
CKAN documentation

Release 2.4.9

Open Knowledge Foundation

September 27, 2017

1	Overview	1
2	User guide	3
2.1	What is CKAN?	3
2.2	Using CKAN	4
3	Sysadmin guide	17
3.1	Creating a sysadmin account	17
3.2	Customizing look and feel	17
3.3	Managing organizations and datasets	18
3.4	Permanently deleting datasets	19
3.5	Managing users	20
4	Maintainer's guide	21
4.1	Installing CKAN	21
4.2	Upgrading CKAN	35
4.3	Getting started	42
4.4	Command Line Interface	43
4.5	Organizations and authorization	50
4.6	Data preview and visualization	53
4.7	FileStore and file uploads	65
4.8	DataStore extension	67
4.9	Apps & Ideas	76
4.10	Tag Vocabularies	77
4.11	Form Integration	78
4.12	Linked Data and RDF	79
4.13	Background tasks	80
4.14	Email notifications	83
4.15	Page View Tracking	84
4.16	Multilingual Extension	87
4.17	Stats Extension	88
4.18	Configuration Options	88
4.19	Multicore Solr setup	113
5	API guide	117
5.1	Legacy APIs	118
5.2	Making an API request	125
5.3	Example: Importing datasets with the CKAN API	127

5.4	API versions	128
5.5	Authentication and API keys	128
5.6	GET-able API functions	129
5.7	JSONP support	129
5.8	API Examples	129
5.9	Action API reference	130
6	Extending guide	165
6.1	Writing extensions tutorial	165
6.2	Using custom config settings in extensions	174
6.3	Making configuration options runtime-editable	175
6.4	Testing extensions	179
6.5	Best practices for writing extensions	183
6.6	Customizing dataset and resource metadata fields using IDatasetForm	184
6.7	Plugin interfaces reference	193
6.8	Plugins toolkit reference	209
6.9	Validator functions reference	215
7	Theming guide	221
7.1	Customizing CKAN's templates	221
7.2	Adding static files	238
7.3	Customizing CKAN's CSS	240
7.4	Adding CSS and JavaScript files using Fanstatic	244
7.5	Customizing CKAN's JavaScript	246
7.6	Best practices for writing CKAN themes	259
7.7	Custom Jinja2 tags reference	262
7.8	Variables and functions available to templates	262
7.9	Objects and methods available to JavaScript modules	264
7.10	Template helper functions reference	265
7.11	Template snippets reference	273
7.12	JavaScript sandbox reference	273
7.13	JavaScript API client reference	273
7.14	CKAN jQuery plugins reference	274
8	Contributing guide	275
8.1	Reporting issues	275
8.2	Translating CKAN	275
8.3	Testing CKAN	279
8.4	Writing commit messages	282
8.5	Making a pull request	282
8.6	Reviewing and merging a pull request	283
8.7	Writing documentation	285
8.8	Projects for beginner CKAN developers	294
8.9	CKAN code architecture	295
8.10	CSS coding standards	298
8.11	HTML coding standards	300
8.12	JavaScript coding standards	303
8.13	Python coding standards	309
8.14	String internationalization	315
8.15	Testing coding standards	319
8.16	Frontend development guidelines	334
8.17	Database migrations	359
8.18	Upgrading CKAN's dependencies	361
8.19	Doing a CKAN release	361

9	Changelog	367
9.1	v2.4.9 2017-09-27	367
9.2	v2.4.8 2017-08-02	367
9.3	v2.4.7 2017-03-22	367
9.4	v2.4.6 2017-02-22	368
9.5	v2.4.5 2017-02-22	368
9.6	v2.4.4 2016-11-02	368
9.7	v2.4.3 2016-03-31	369
9.8	v2.4.2 2015-12-17	369
9.9	v2.4.1 2015-09-02	369
9.10	v2.4.0 2015-07-22	369
9.11	v2.3.5 2016-11-02	371
9.12	v2.3.4 2016-03-31	371
9.13	v2.3.3 2015-12-17	372
9.14	v2.3.2 2015-09-02	372
9.15	v2.3.1 2015-07-22	372
9.16	v2.3 2015-03-04	372
9.17	v2.2.4 2015-12-17	378
9.18	v2.2.3 2015-07-22	378
9.19	v2.2.2 2015-03-04	379
9.20	v2.2.1 2014-10-15	379
9.21	v2.2 2014-02-04	380
9.22	v2.1.6 2015-12-17	384
9.23	v2.1.5 2015-07-22	384
9.24	v2.1.4 2015-03-04	384
9.25	v2.1.3 2014-10-15	384
9.26	v2.1.2 2014-02-04	385
9.27	v2.1.1 2013-11-8	386
9.28	v2.1 2013-08-13	386
9.29	v2.0.8 2015-12-17	388
9.30	v2.0.7 2015-07-22	388
9.31	v2.0.6 2015-03-04	388
9.32	v2.0.5 2014-10-15	389
9.33	v2.0.4 2014-02-04	389
9.34	v2.0.3 2013-11-8	389
9.35	v2.0.2 2013-08-13	390
9.36	v2.0.1 2013-06-11	390
9.37	v2.0 2013-05-10	390
9.38	v1.8.2 2013-08-13	393
9.39	v1.8.1 2013-05-10	393
9.40	v1.8 2012-10-19	394
9.41	v1.7.4 2013-08-13	395
9.42	v1.7.3 2013-05-10	395
9.43	v1.7.2 2012-10-19	395
9.44	v1.7.1 2012-06-20	395
9.45	v1.7 2012-05-09	396
9.46	v1.6 2012-02-24	397
9.47	v1.5.1 2012-01-04	398
9.48	v1.5 2011-11-07	399
9.49	v1.4.3.1 2011-09-30	400
9.50	v1.4.3 2011-09-13	400
9.51	v1.4.2 2011-08-05	401
9.52	v1.4.1 2011-06-27	402
9.53	v1.4 2011-05-19	402

9.54	v1.3.3 2011-04-08	402
9.55	v1.3.2 2011-03-15	403
9.56	v1.3 2011-02-18	404
9.57	v1.2 2010-11-25	404
9.58	v1.1 2010-08-10	405
9.59	v1.0.2 2010-08-27	405
9.60	v1.0.1 2010-06-23	405
9.61	v1.0 2010-05-11	405
9.62	v0.11 2010-01-25	406
9.63	v0.10 2009-09-30	407
9.64	v0.9 2009-07-31	407
9.65	v0.8 2009-04-10	408
9.66	v0.7 2008-10-31	408
9.67	v0.6 2008-07-08	408
9.68	v0.5 2008-01-22	408
9.69	v0.4 2007-07-04	409
9.70	v0.3 2007-04-12	409
9.71	v0.2 2007-02	409
9.72	v0.1 2006-05	409

Python Module Index	411
----------------------------	------------

Overview

Welcome to CKAN's documentation! These docs are organized into several guides, each guide serves a different audience of CKAN users or contributors:

User guide The guide for people who will be using a CKAN site through its web interface, explains how to use the web interface and its features.

Sysadmin guide The guide for people with a sysadmin account on a CKAN site, explains how to use the sysadmin features of the CKAN web interface.

Maintainer's guide The guide for people who will be maintaining a CKAN site, explains how to install, upgrade and configure CKAN and its features and extensions.

API guide The guide for API developers, explains how to write code that interacts with CKAN sites and their data.

Extending guide The guide for extension developers, explains how to customize and extend CKAN's features.

Theming guide The guide for theme developers, explains how to customize the appearance and content of CKAN's web interface.

Contributing guide The guide for people who want to contribute to CKAN itself, explains how to make all kinds of contributions to CKAN, including reporting bugs, testing and QA, translating CKAN, helping with CKAN's documentation, and contributing to the CKAN code.

Finally, the ***Changelog*** documents the differences between CKAN releases, useful information for anyone who is using CKAN.

Note: These docs are maintained by the CKAN development team. CKAN and its documentation are free and open source, and contributions are welcome. To contribute to these docs, see *[Writing documentation](#)*.

User guide

This user guide covers using CKAN's web interface to organize, publish and find data. CKAN also has a powerful API (machine interface), which makes it easy to develop extensions and links with other information systems. The API is documented in [API guide](#).

Some web UI features relating to site administration are available only to users with sysadmin status, and are documented in [Sysadmin guide](#).

What is CKAN?

CKAN is a tool for making open data websites. (Think of a content management system like WordPress - but for data, instead of pages and blog posts.) It helps you manage and publish collections of data. It is used by national and local governments, research institutions, and other organizations who collect a lot of data.

Once your data is published, users can use its faceted search features to browse and find the data they need, and preview it using maps, graphs and tables - whether they are developers, journalists, researchers, NGOs, citizens, or even your own staff.

Datasets and resources

For CKAN purposes, data is published in units called “datasets”. A dataset is a parcel of data - for example, it could be the crime statistics for a region, the spending figures for a government department, or temperature readings from various weather stations. When users search for data, the search results they see will be individual datasets.

A dataset contains two things:

- Information or “metadata” about the data. For example, the title and publisher, date, what formats it is available in, what license it is released under, etc.
- A number of “resources”, which hold the data itself. CKAN does not mind what format the data is in. A resource can be a CSV or Excel spreadsheet, XML file, PDF document, image file, linked data in RDF format, etc. CKAN can store the resource internally, or store it simply as a link, the resource itself being elsewhere on the web. A dataset can contain any number of resources. For example, different resources might contain the data for different years, or they might contain the same data in different formats.

Users, organizations and authorization

CKAN users can register user accounts and log in. Normally (depending on the site setup), login is not needed to search for and find data, but is needed for all publishing functions: datasets can be created, edited, etc by users with the appropriate permissions.

Normally, each dataset is owned by an “organization”. A CKAN instance can have any number of organizations. For example, if CKAN is being used as a data portal by a national government, the organizations might be different government departments, each of which publishes data. Each organization can have its own workflow and authorizations, allowing it to manage its own publishing process.

An organization’s administrators can add individual users to it, with different roles depending on the level of authorization needed. A user in an organization can create a dataset owned by that organization. In the default setup, this dataset is initially private, and visible only to other users in the same organization. When it is ready for publication, it can be published at the press of a button. This may require a higher authorization level within the organization.

Datasets cannot normally be created except within organizations. It is possible, however, to set up CKAN to allow datasets not owned by any organization. Such datasets can be edited by any logged-in user, creating the possibility of a wiki-like datahub.

Note: The user guide covers all the main features of the web user interface (UI). In practice, depending on how the site is configured, some of these functions may be slightly different or unavailable. For example, in an official CKAN instance in a production setting, the site administrator will probably have made it impossible for users to create new organizations via the UI. You can try out all the features described at <http://demo.ckan.org>.

Using CKAN

Registering and logging in

Note: Registration is needed for most publishing features and for personalization features, such as “following” datasets. It is not needed to search for and download data.

To create a user ID, use the “Register” link at the top of any page. CKAN will ask for the following:

- *Username* – choose a username using only letters, numbers, - and _ characters. For example, “jbloggs” or “joe_bloggs93”.
- *Full name* – to be displayed on your user profile
- *E-mail address* – this will not be visible to other users
- *Password* – enter the same password in both boxes

If there are problems with any of the fields, CKAN will tell you the problem and enable you to correct it. When the fields are filled in correctly, CKAN will create your user account and automatically log you in.

Note: It is perfectly possible to have more than one user account attached to the same e-mail address. For this reason, choose a username you will remember, as you will need it when logging in.

Features for publishers

Adding a new dataset

Note: You may need to be a member of an organization in order to add and edit datasets. See the section [Creating an organization](#) below. On <http://demo.ckan.org>, you can add a dataset without being in an organization, but dataset features relating to authorization and organizations will not be available.

Step 1. You can access CKAN's "Create dataset" screen in two ways.

1. Select the "Datasets" link at the top of any page. From this, above the search box, select the "Add Dataset" button.
2. Alternatively, select the "organizations" link at the top of a page. Now select the page for the organization that should own your new dataset. Provided that you are a member of this organization, you can now select the "Add Dataset" button above the search box.

Step 2. CKAN will ask for the following information about your data. (The actual data will be added in step 4.)

- *Title* – this title will be unique across CKAN, so make it brief but specific. E.g. "UK population density by region" is better than "Population figures".

- *Description* – You can add a longer description of the dataset here, including information such as where the data is from and any information that people will need to know when using the data.
- *Tags* – here you may add tags that will help people find the data and link it with other related data. Examples could be “population”, “crime”, “East Anglia”. Hit the <return> key between tags. If you enter a tag wrongly, you can use its delete button to remove it before saving the dataset.
- *License* – it is important to include license information so that people know how they can use the data. This field should be a drop-down box. If you need to use a license not on the list, contact your site administrator.
- *Organization* - If you are a member of any organizations, this drop-down will enable you to choose which one should own the dataset. Ensure the default chosen is the correct one before you proceed. (Probably most users will be in only one organization. If this is you, CKAN will have chosen your organization by default and you need not do anything.)

The screenshot shows the CKAN 'Create Dataset' form. The browser title is 'Create dataset - CKAN'. The CKAN logo is in the top left, and navigation links for 'Datasets', 'Organizations', 'Groups', and 'About' are in the top right. A search bar is also present. The main heading is '/ Datasets / Create Dataset'. Below this is a progress bar with three steps: '1 Create dataset' (highlighted in green), '2 Add data', and '3 Additional data'. The form fields are as follows:

- Title:** UP Library catalogue
- URL:** /dataset/up-library-catalogue
- Description:** List of books held in Upper Pagwell Village Library
- Tags:** library, Pagwell, bibliography
- License:** Creative Commons Attribution...
- Organization:** pagwell-borough-council

At the bottom right, there are 'Cancel' and 'Next: Add Data' buttons. A sidebar on the left contains a 'What are datasets?' section with a brief explanation of datasets.

Note: By default, the only required field on this page is the title. However, it is good practice to include, at the minimum, a short description and, if possible, the license information. You should ensure that you choose the correct organization for the dataset, since at present, this cannot be changed later. You can edit or add to the other fields later.

Step 3. When you have filled in the information on this page, select the “Next: Add Data” button. (Alternatively select “Cancel” to discard the information filled in.)

Step 4. CKAN will display the “Add data” screen.

This is where you will add one or more “resources” which contain the data for this dataset. Choose a file or link for your data resource and select the appropriate choice at the top of the screen:

- If you are giving CKAN a link to the data, like `http://example.com/mydata.csv`, then select “Link to a file” or “Link to an API”. (If you don’t know what an API is, you don’t need to worry about this option - select “Link to a file”.)
- If the data to be added to CKAN is in a file on your computer, select “Upload a file”. CKAN will give you a file browser to select it.

Step 5. Add the other information on the page. CKAN does not require this information, but it is good practice to add it:

- *Name* – a name for this resource, e.g. “Population density 2011, CSV”. Different resources in the dataset should have different names.
- *Description* – a short description of the resource.
- *Format* – the file format of the resource, e.g. CSV (comma-separated values), XLS, JSON, PDF, etc.

Step 6. If you have more resources (files or links) to add to the dataset, select the “Save & add another” button. When you have finished adding resources, select “Next: Additional Info”.

Step 7. CKAN displays the “Additional data” screen.

- *Visibility* – a `Public` dataset is public and can be seen by any user of the site. A `Private` dataset can only be seen by members of the organization owning the dataset and will not show up in searches by other users.
- *Author* – The name of the person or organization responsible for producing the data.
- *Author e-mail* – an e-mail address for the author, to which queries about the data should be sent.
- *Maintainer / maintainer e-mail* – If necessary, details for a second person responsible for the data.

- *Custom fields* – If you want the dataset to have another field, you can add the field name and value here. E.g. “Year of publication”. Note that if there is an extra field that is needed for a large number of datasets, you should talk to your site administrator about changing the default schema and dataset forms.

Note: Everything on this screen is optional, but you should ensure the “Visibility” is set correctly. It is also good practice to ensure an Author is named.

Changed in version 2.2: Previous versions of CKAN used to allow adding the dataset to existing groups in this step. This was changed. To add a dataset to an existing group now, go to the “Group” tab in the Dataset’s page.

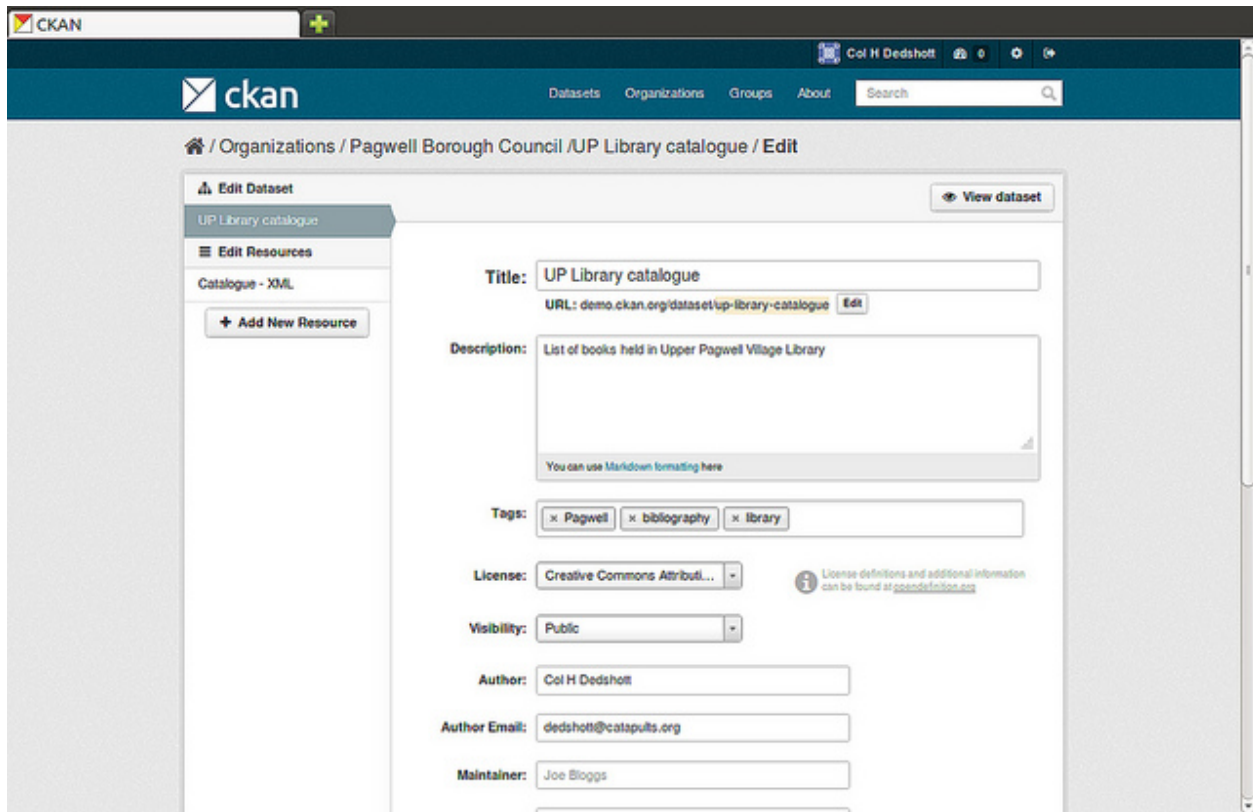
Step 8. Select the ‘Finish’ button. CKAN creates the dataset and shows you the result. You have finished!

You should be able to find your dataset by typing the title, or some relevant words from the description, into the search box on any page in your CKAN instance. For more information about finding data, see the section [Finding data](#).

Editing a dataset

You can edit the dataset you have created, or any dataset owned by an organization that you are a member of. (If a dataset is not owned by any organization, then any registered user can edit it.)

1. Go to the dataset’s page. You can find it by entering the title in the search box on any page.
2. Select the “Edit” button, which you should see above the dataset title.
3. CKAN displays the “Edit dataset” screen. You can edit any of the fields (Title, Description, Dataset, etc), change the visibility (Private/Public), and add or delete tags or custom fields. For details of these fields, see [Adding a new dataset](#).
4. When you have finished, select the “Update dataset” button to save your changes.



Adding, deleting and editing resources

1. Go to the dataset's "Edit dataset" page (steps 1-2 above).
2. In the left sidebar, there are options for editing resources. You can select an existing resource (to edit or delete it), or select "Add new resource".
3. You can edit the information about the resource or change the linked or uploaded file. For details, see steps 4-5 of "Adding a new resource", above.
4. When you have finished editing, select the button marked "Update resource" (or "Add", for a new resource) to save your changes. Alternatively, to delete the resource, select the "Delete resource" button.

Deleting a dataset

1. Go to the dataset's "Edit dataset" page (see "Editing a dataset", above).
2. Select the "Delete" button.
3. CKAN displays a confirmation dialog box. To complete deletion of the dataset, select "Confirm".

Note: The "Deleted" dataset is not completely deleted. It is hidden, so it does not show up in any searches, etc. However, by visiting the URL for the dataset's page, it can still be seen (by users with appropriate authorization), and "undeleted" if necessary. If it is important to completely delete the dataset, contact your site administrator.

Creating an organization

In general, each dataset is owned by one organization. Each organization includes certain users, who can modify its datasets and create new ones. Different levels of access privileges within an organization can be given to users, e.g. some users might be able to edit datasets but not create new ones, or to create datasets but not publish them. Each organization has a home page, where users can find some information about the organization and search within its datasets. This allows different data publishing departments, bodies, etc to control their own publishing policies.

To create an organization:

1. Select the “Organizations” link at the top of any page.
2. Select the “Add Organization” button below the search box.
3. CKAN displays the “Create an Organization” page.
4. Enter a name for the organization, and, optionally, a description and image URL for the organization’s home page.
5. Select the “Create Organization” button. CKAN creates your organization and displays its home page. Initially, of course, the organization has no datasets.

The screenshot shows the CKAN web interface for creating a new organization. The browser tab is 'Create an Organization - CKAN'. The user is logged in as 'Col H Dedshott'. The navigation bar includes links for 'Datasets', 'Organizations', 'Groups', and 'About', along with a search bar. The breadcrumb trail is 'Home / Organizations / Create an Organization'. The main content area is titled 'Create an Organization' and contains a form with the following fields:

- Title:** Pagwell Borough Council
- URL:** demo.ckan.org/organization/pagwell-borough-council (with an 'Edit' button)
- Description:** Open data from Great Pagwell, Little Pagwell, Pagwell Magna, Pagwell Parva, Pagwell-on-Sea and Upper and Lower Pagwells. (with a 'You can use Markdown formatting here' note)
- Image URL:** http://pagwell.gov.uk/images/pagwell.jpg

A 'Create Organization' button is located at the bottom right of the form. On the left side, there is a sidebar with two sections: 'What are Organizations?' and 'Info'. The 'What are Organizations?' section explains that organizations act like publishing departments for datasets and that admins can assign roles to members. The 'Info' section is currently empty.

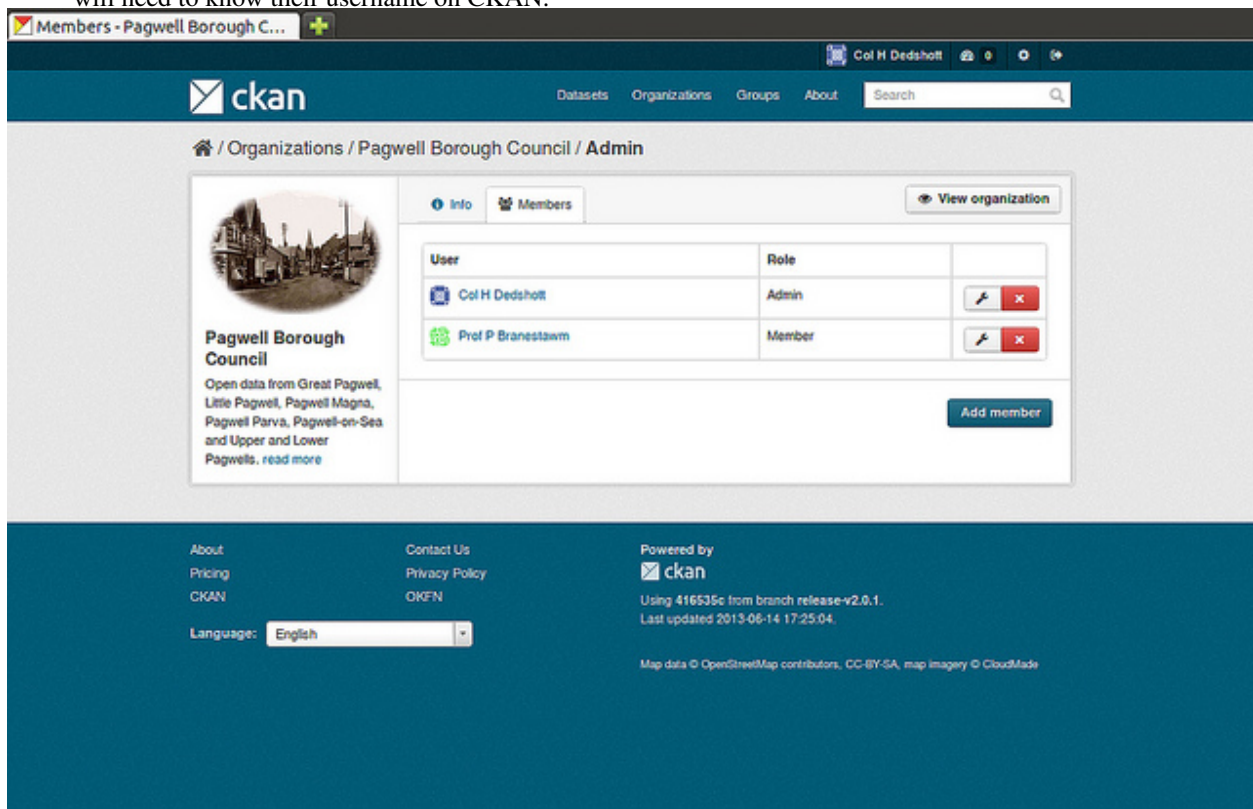
You can now change the access privileges to the organization for other users - see [Managing an organization](#) below. You can also create datasets owned by the organization; see [Adding a new dataset](#) above.

Note: Depending on how CKAN is set up, you may not be authorized to create new organizations. In this case, if you need a new organization, you will need to contact your site administrator.

Managing an organization

When you create an organization, CKAN automatically makes you its “Admin”. From the organization’s page you should see an “Admin” button above the search box. When you select this, CKAN displays the organization admin page. This page has two tabs:

- *Info* – Here you can edit the information supplied when the organization was created (title, description and image).
- *Members* – Here you can add, remove and change access roles for different users in the organization. Note: you will need to know their username on CKAN.



By default CKAN allows members of organizations with three roles:

- *Member* – can see the organization’s private datasets
- *Editor* – can edit and publish datasets
- *Admin* – can add, remove and change roles for organization members

Finding data

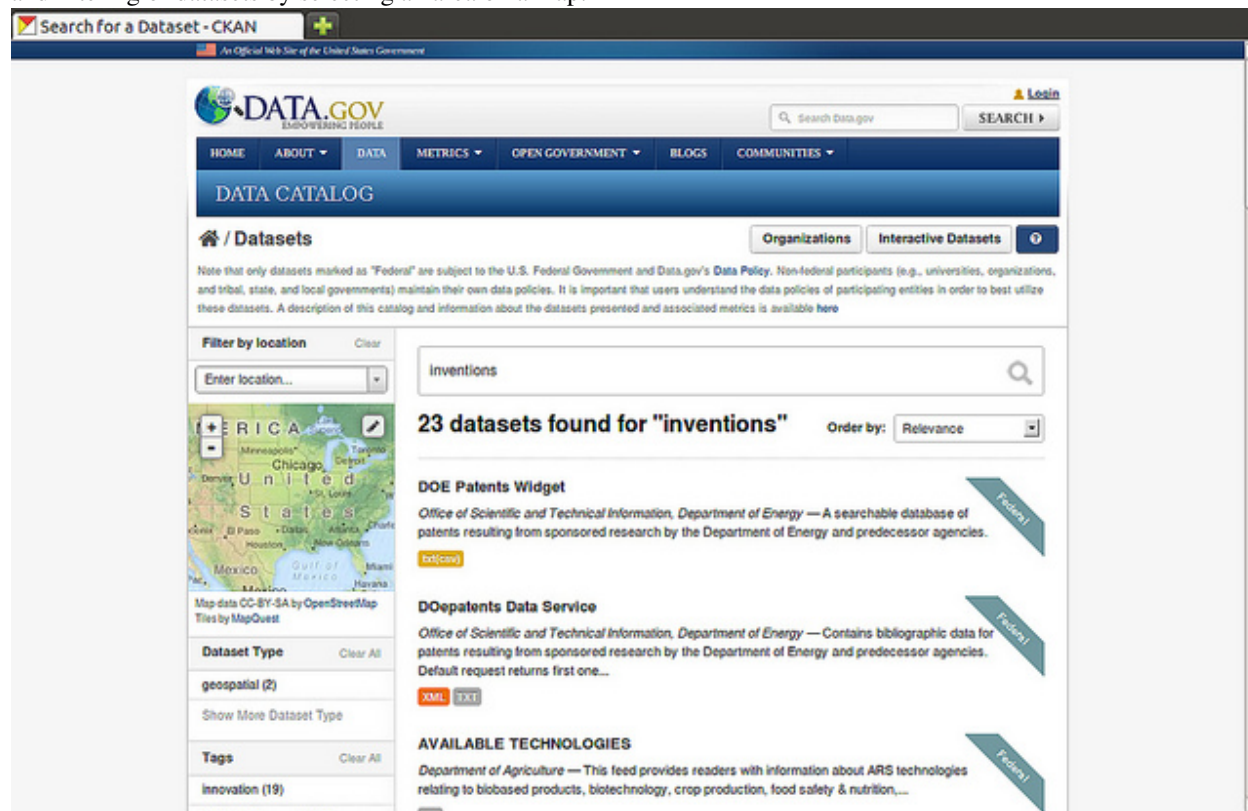
Searching the site

To find datasets in CKAN, type any combination of search words (e.g. “health”, “transport”, etc) in the search box on any page. CKAN displays the first page of results for your search. You can:

- View more pages of results
- Repeat the search, altering some terms
- Restrict the search to datasets with particular tags, data formats, etc using the filters in the left-hand column

If there are a large number of results, the filters can be very helpful, since you can combine filters, selectively adding and removing them, and modify and repeat the search with existing filters still in place.

If datasets are tagged by geographical area, it is also possible to run CKAN with an extension which allows searching and filtering of datasets by selecting an area on a map.



Searching within an organization

If you want to look for data owned by a particular organization, you can search within that organization from its home page in CKAN.

1. Select the “Organizations” link at the top of any page.
2. Select the organization you are interested in. CKAN will display your organization’s home page.
3. Type your search query in the main search box on the page.

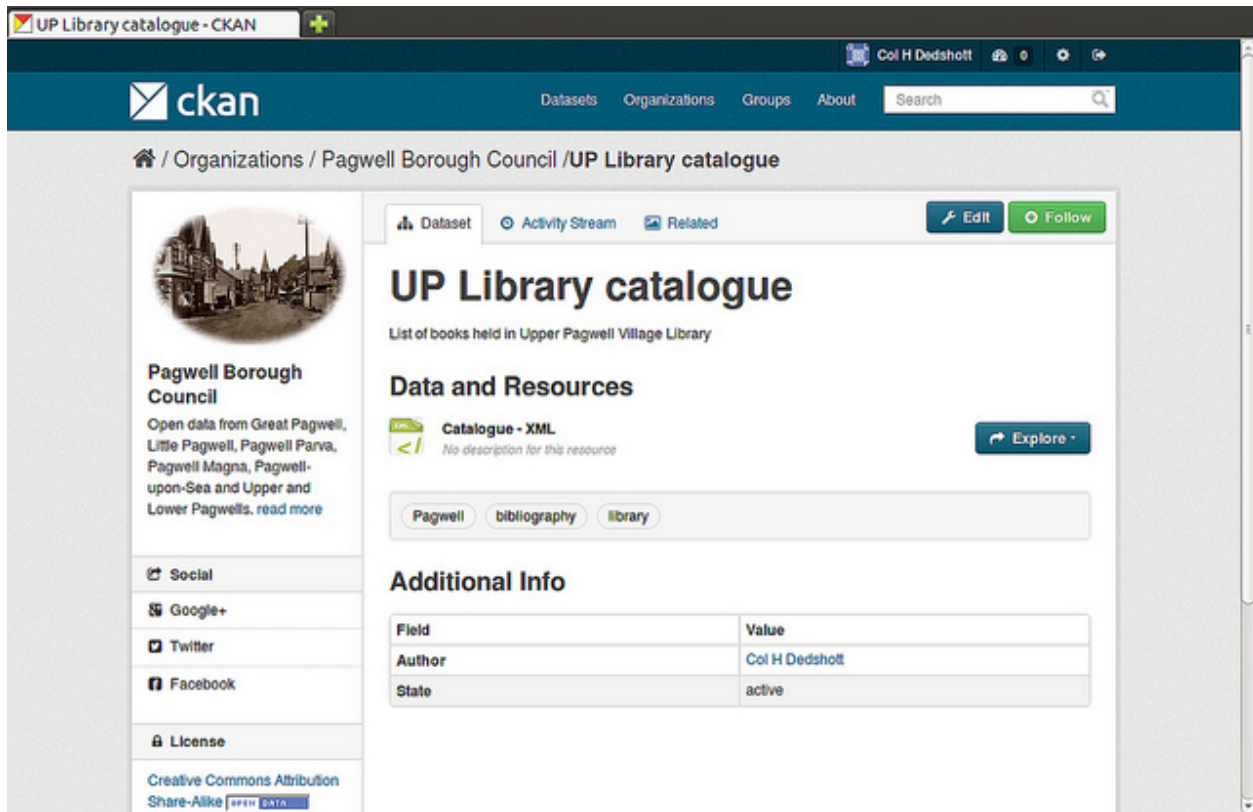
CKAN will return search results as normal, but restricted to datasets from the organization.

If the organization is of interest, you can opt to be notified of changes to it (such as new datasets and modifications to datasets) by using the “Follow” button on the organization page. See the section *Managing your news feed* below. You must have a user account and be logged in to use this feature.

Exploring datasets

When you have found a dataset you are interested and selected it, CKAN will display the dataset page. This includes

- The name, description, and other information about the dataset
- Links to and brief descriptions of each of the resources



The resource descriptions link to a dedicated page for each resource. This resource page includes information about the resource, and enables it to be downloaded. Many types of resource can also be previewed directly on the resource page. .CSV and .XLS spreadsheets are previewed in a grid view, with map and graph views also available if the data is suitable. The resource page will also preview resources if they are common image types, PDF, or HTML.

The dataset page also has two other tabs:

- *Activity stream* – see the history of recent changes to the dataset
- *Related items* – see any links to web pages related to this dataset, or add your own links.

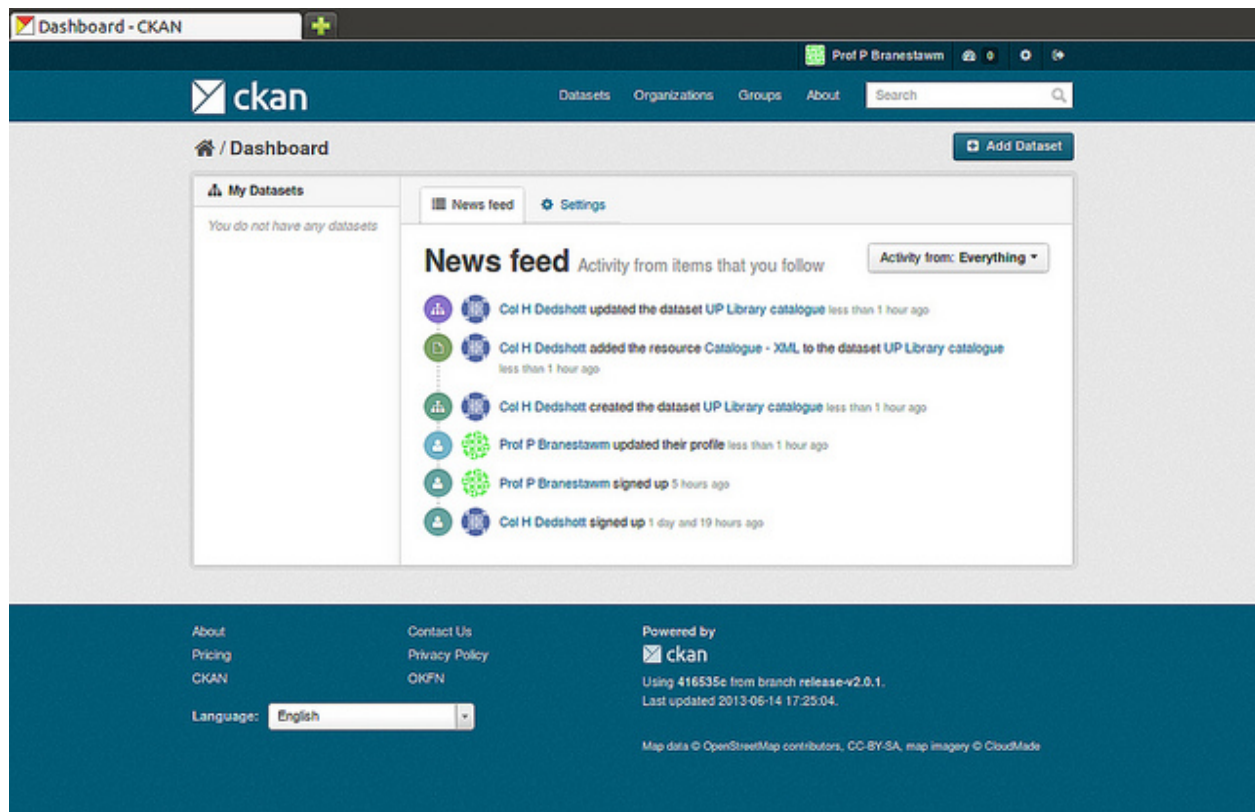
If the dataset is of interest, you can opt to be notified of changes to it by using the “Follow” button on the dataset page. See the section [Managing your news feed](#) below. You must have a user account and be logged in to use this feature.

Personalization

CKAN provides features to personalize the experience of both searching for and publishing data. You must be logged in to use these features.

Managing your news feed

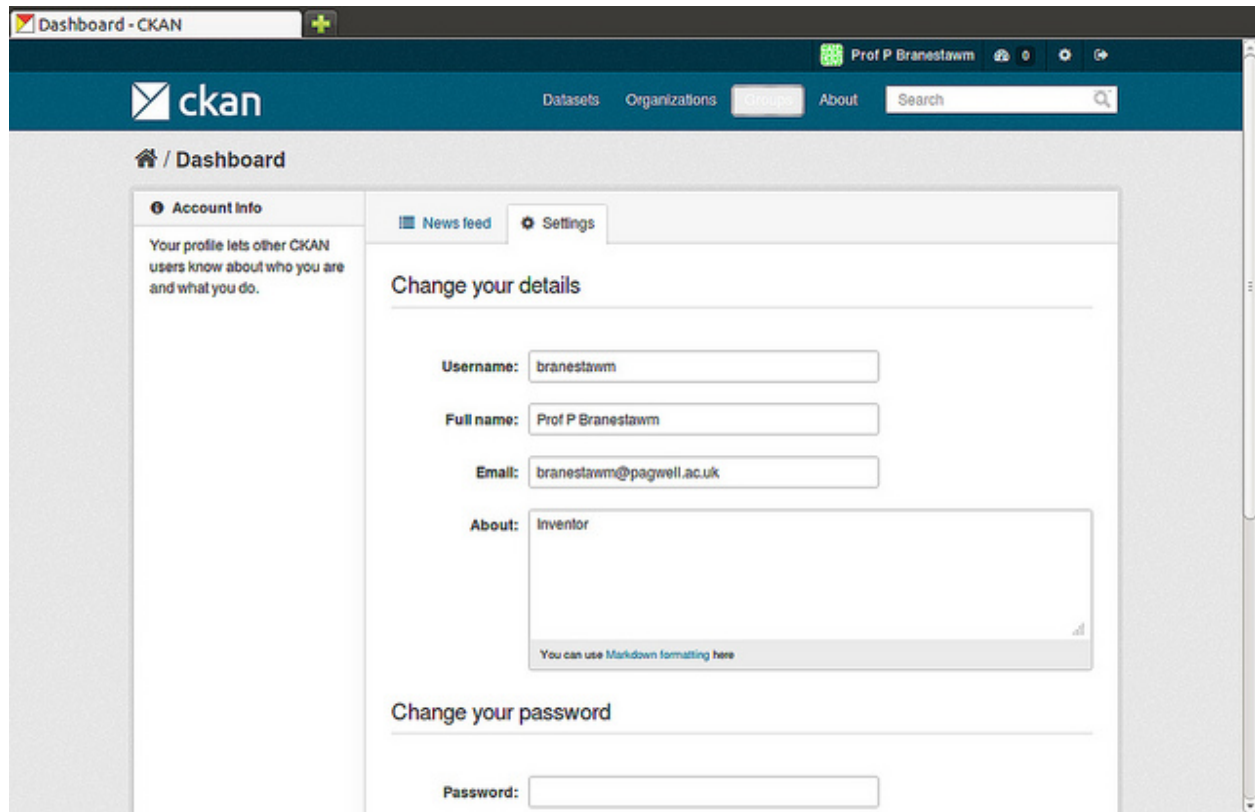
At the top of any page, select the dashboard symbol (next to your name). CKAN displays your News feed. This shows changes to datasets that you follow, and any changed or new datasets in organizations that you follow. The number by the dashboard symbol shows the number of new notifications in your News feed since you last looked at it. As well as datasets and organizations, it is possible to follow individual users (to be notified of changes that they make to datasets).



If you want to stop following a dataset (or organization or user), go to the dataset's page (e.g. by selecting a link to it in your News feed) and select the “Unfollow” button.

Managing your user profile

You can change the information that CKAN holds about you, including what other users see about you by editing your user profile. (Users are most likely to see your profile when you edit a dataset or upload data to an organization that they are following.) To do this, select the gearwheel symbol at the top of any page.



The screenshot shows the CKAN user settings page. The browser address bar displays 'Dashboard - CKAN'. The CKAN logo is in the top left, and navigation links for 'Datasets', 'Organizations', 'Groups', 'About', and a search bar are in the top right. The user 'Prof P Branestawm' is logged in. The page title is 'Dashboard'. On the left, the 'Account Info' section explains that the profile lets other CKAN users know about the user. The main content area has tabs for 'News feed' and 'Settings'. Under 'Settings', there are two sections: 'Change your details' and 'Change your password'. The 'Change your details' section contains input fields for 'Username' (branestawm), 'Full name' (Prof P Branestawm), 'Email' (branestawm@pagwell.ac.uk), and 'About' (Inventor). A note at the bottom of the 'About' field states 'You can use Markdown formatting here'. The 'Change your password' section has a 'Password' input field.

CKAN displays the user settings page. Here you can change:

- Your username
- Your full name
- Your e-mail address (note: this is not displayed to other users)
- Your profile text - an optional short paragraph about yourself
- Your password

Make the changes you require and then select the “Update Profile” button.

Note: If you change your username, CKAN will log you out. You will need to log back in using your new username.

Sysadmin guide

This guide covers the administration features of CKAN 2.0, such as managing users and datasets. These features are available via the web user interface to a user with sysadmin rights. The guide assumes familiarity with the [User guide](#).

Certain administration tasks are not available through the web UI but need access to the server where CKAN is installed. These include the range of configuration options using the site's "config" file, documented in [Configuration Options](#), and those available via [Command Line Interface](#).

Warning: A sysadmin user can access and edit any organizations, view and change user details, and permanently delete datasets. You should carefully consider who has access to a sysadmin account on your CKAN system.

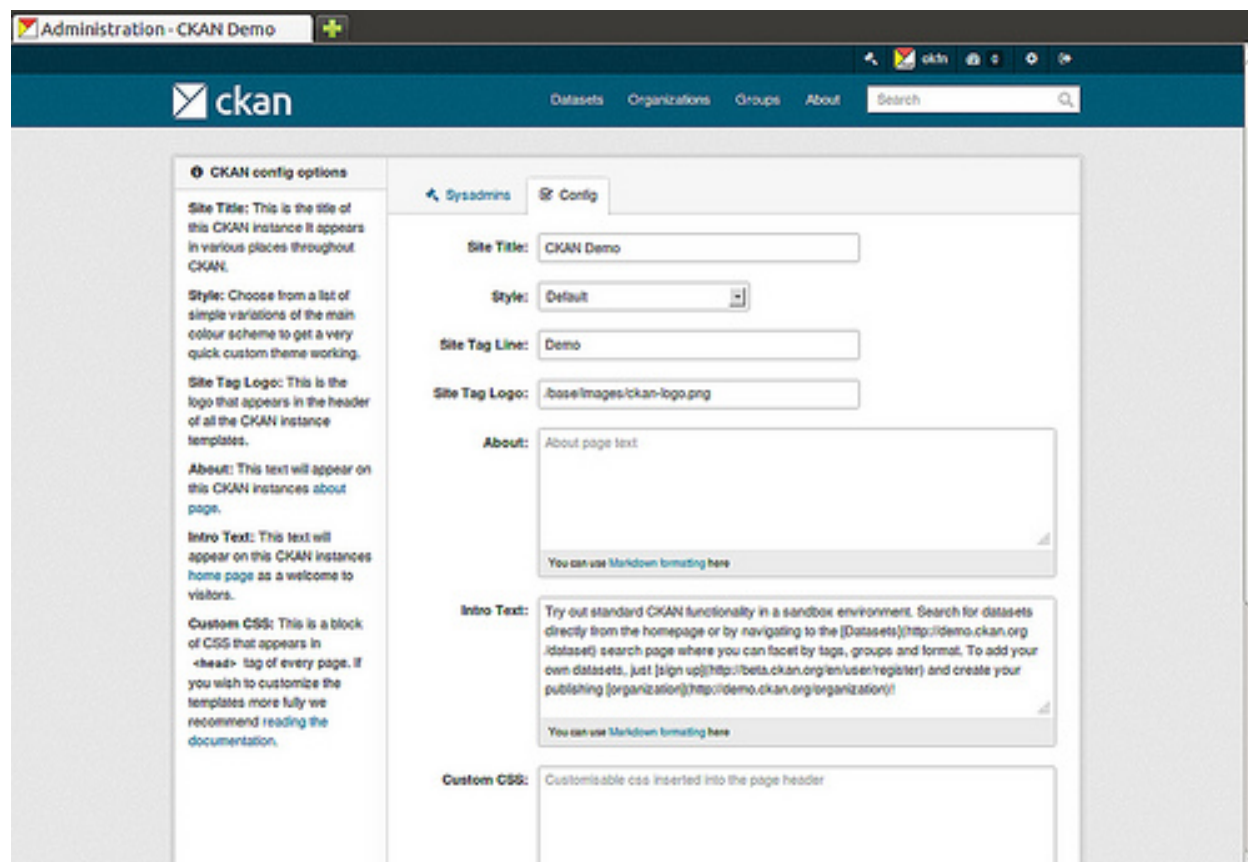
Creating a sysadmin account

Normally, a sysadmin account is created as part of the process of setting up CKAN. If one does not already exist, you will need to create a sysadmin user, or give sysadmin rights to an existing user. To do this requires access to the server; see [Creating a sysadmin user](#) for details. If another organization is hosting CKAN, you will need to ask them to create a sysadmin user.

Adding more sysadmin accounts is done in the same way. It cannot be done via the web UI.

Customizing look and feel

Some simple customizations to customize the 'look and feel' of your CKAN site are available via the UI, at `http://<my-ckan-url>/ckan-admin/config/`.



Here you can edit the following:

Site title This title is used in the HTML `<title>` of pages served by CKAN (which may be displayed on your browser's title bar). For example if your site title is "CKAN Demo", the home page is called "Welcome - CKAN Demo". The site title is also used in a few other places, e.g. in the alt-text of the main site logo.

Style Choose one of five colour schemes for the default theme.

Site tag line This is not used in CKAN's current default themes, but may be used in future.

Site tag logo A URL for the site logo, used at the head of every page of CKAN.

About Text that appears on the "about" page, <http://<my-ckan-url>/about>. You can use [Markdown](#) here. If it is left empty, a standard text describing CKAN will appear.

Intro text This text appears prominently on the home page of your site.

Custom CSS For simple style changes, you can add CSS code here which will be added to the `<head>` of every page.

Managing organizations and datasets

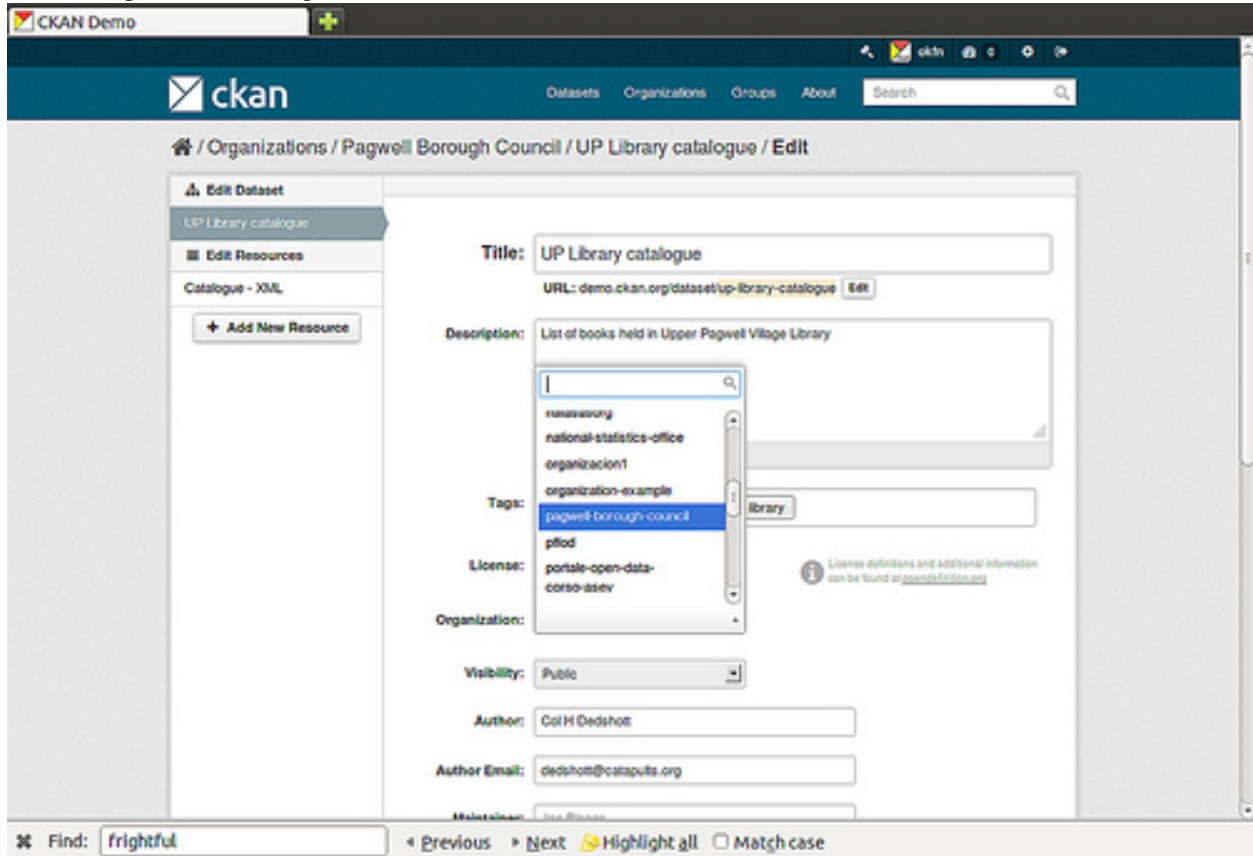
A sysadmin user has full access to user accounts, organizations and datasets. For example, you have access to every organization as if you were a member of that organization. Thus most management operations are done in exactly the same way as in the normal web interface.

For example, to add or delete users to an organization, change a user's role in the organization, delete the organization or edit its description, etc, visit the organization's home page. You will see the 'Admin' button as if you were a member of the organization. You can use this to perform all organization admin functions. For details, see the [User guide](#).

Similarly, to edit, update or delete a dataset, go to the dataset page and use the ‘Edit’ button. As an admin user you can see all datasets including those that are private to an organization. They will show up when doing a dataset search.

Moving a dataset between organizations

To move a dataset between organizations, visit the dataset’s Edit page. Choose the appropriate entry from the “organization” drop-down list, and press “Save”.



Permanently deleting datasets

A dataset which has been deleted is not permanently removed from CKAN; it is simply marked as ‘deleted’ and will no longer show up in search, etc. The dataset’s URL cannot be re-used for a new dataset.

To permanently delete (“purge”) a dataset:

- Navigate to the dataset’s “Edit” page, and delete it.
- Visit `http://<my-ckan-url>/ckan-admin/trash/`.

This page shows all deleted datasets and allows you to delete them permanently.

Warning: This operation cannot be reversed!

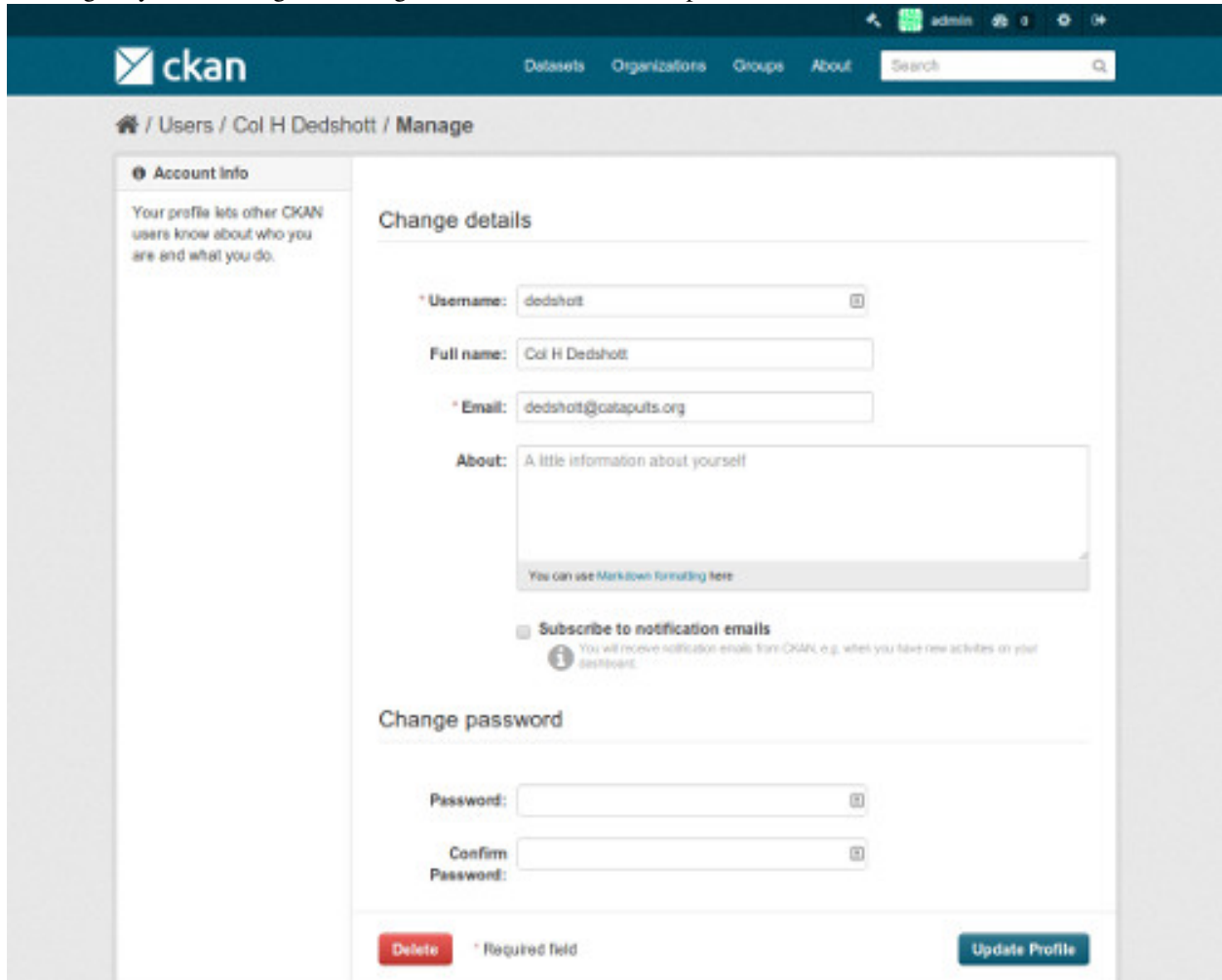
Note: At present, it is not possible to purge organizations or groups using the web UI. This can only be done with access to the server, by directly deleting them from CKAN’s database.

Managing users

To find a user's profile, go to `http://<my-ckan-url>/user/`. You can search for users in the search box provided.

You can search by any part of the user profile, including their e-mail address. This is useful if, for example, a user has forgotten their user ID. For non-sysadmin users, the search on this page will only match public parts of the profile, so they cannot search by e-mail address.

On their user profile, you will see a “Manage” button. CKAN displays the user settings page. You can delete the user or change any of its settings, including their username, name and password.



The screenshot shows the CKAN user profile management interface. The top navigation bar includes the CKAN logo, links for Datasets, Organizations, Groups, and About, and a search box. The breadcrumb trail indicates the user is viewing the profile for 'Col H Dedshott' and is in the 'Manage' section. The main content area is divided into two columns. The left column, titled 'Account info', contains a message: 'Your profile lets other CKAN users know about who you are and what you do.' The right column, titled 'Change details', contains several form fields: 'Username' (with value 'dedshott'), 'Full name' (with value 'Col H Dedshott'), 'Email' (with value 'dedshott@catapults.org'), and 'About' (a text area with placeholder text 'A little information about yourself' and a note 'You can use Markdown formatting here'). Below these fields is a checkbox for 'Subscribe to notification emails' with a tooltip that says 'You will receive notification emails from CKAN, e.g. when you have new activities on your dashboard.' At the bottom of the 'Change details' section is a 'Change password' section with 'Password' and 'Confirm Password' fields. At the very bottom, there is a red 'Delete' button, a note '* Required field', and a blue 'Update Profile' button.

New in version 2.2: Previous versions of CKAN didn't allow you to delete users through the web interface.

Maintainer's guide

The sections below document how to setup and maintain a CKAN site, including installing, upgrading and configuring CKAN and its features and extensions.

Installing CKAN

Before you can use CKAN on your own computer, you need to install it. There are three ways to install CKAN:

1. Install from an operating system package
2. Install from source
3. Install using a [Docker](#) image

From package is the quickest and easiest way to install CKAN, but it requires Ubuntu 12.04 64-bit. **You should install CKAN from package if:**

- You want to install CKAN on an Ubuntu 12.04, 64-bit server, *and*
- You only want to run one CKAN website per server

See [Installing CKAN from package](#).

You should install CKAN from source if:

- You want to install CKAN on a 32-bit computer, *or*
- You want to install CKAN on a different version of Ubuntu, not 12.04, *or*
- You want to install CKAN on another operating system (eg. RedHat, CentOS, OS X), *or*
- You want to run multiple CKAN websites on the same server, *or*
- You want to install CKAN for development

See [Installing CKAN from source](#).

You should install using Docker if:

- You want to deploy CKAN on any server that can run Docker, regardless of operating system
- You want a deployment mechanism that remains the same as you move from vanilla CKAN to a heavily customised deployment

See [Installing CKAN using a Docker image](#).

If you've already setup a CKAN website and want to upgrade it to a newer version of CKAN, see [Upgrading CKAN](#).

Installing CKAN from package

This section describes how to install CKAN from package. This is the quickest and easiest way to install CKAN, but it requires **Ubuntu 12.04 64-bit**. If you're not using Ubuntu 12.04 64-bit, or if you're installing CKAN for development, you should follow *Installing CKAN from source* instead.

At the end of the installation process you will end up with two running web applications, CKAN itself and the Data-Pusher, a separate service for automatically importing data to CKAN's *DataStore extension*.

1. Install the CKAN package

On your Ubuntu 12.04 system, open a terminal and run these commands to install CKAN:

1. Update Ubuntu's package index:

```
sudo apt-get update
```

2. Install the Ubuntu packages that CKAN requires:

```
sudo apt-get install -y nginx apache2 libapache2-mod-wsgi libpq5
```

3. Download the CKAN package:

```
wget http://packaging.ckan.org/python-ckan_2.7_amd64.deb
```

Note: If `wget` is not present, you can install it via:

```
sudo apt-get install wget
```

4. Install the CKAN package:

```
sudo dpkg -i python-ckan_2.7_amd64.deb
```

Note: If you get the following error it means that for some reason the Apache WSGI module was not enabled:

```
Syntax error on line 1 of /etc/apache2/sites-enabled/ckan_default:
Invalid command 'WSGISocketPrefix', perhaps misspelled or defined by a module not included in the server
Action 'configtest' failed.
The Apache error log may have more information.
...fail!
```

You can enable it by running these commands in a terminal:

```
sudo a2enmod wsgi
sudo service apache2 restart
```

2. Install PostgreSQL and Solr

Tip: You can install PostgreSQL, Solr and CKAN on different servers. Just change the *sqlalchemy.url* and *solr_url* settings in your `/etc/ckan/default/production.ini` file to reference your PostgreSQL and Solr servers.

1. Install PostgreSQL and Solr, run this command in a terminal:

```
sudo apt-get install -y postgresql solr-jetty
```

The install will whirr away, then towards the end you'll see this:

* Not starting jetty - edit `/etc/default/jetty` and change `NO_START` to be 0 (or comment it out).

2. Follow the instructions in [5. Setup Solr](#) to setup Solr.
3. Follow the instructions in [3. Setup a PostgreSQL database](#) to setup PostgreSQL, then edit the `sqlalchemy.url` option in your `/etc/ckan/default/production.ini` file and set the correct password, database and database user.

3. Update the configuration and initialize the database

1. Edit the *CKAN configuration file* (`/etc/ckan/default/production.ini`) to set up the following options:

site_id Each CKAN site should have a unique `site_id`, for example:

```
ckan.site_id = default
```

site_url Provide the site's URL. For example:

```
ckan.site_url = http://demo.ckan.org
```

2. Initialize your CKAN database by running this command in a terminal:

```
sudo ckan db init
```
3. Optionally, setup the DataStore and DataPusher by following the instructions in [DataStore extension](#).
4. Also optionally, you can enable file uploads by following the instructions in [FileStore and file uploads](#).

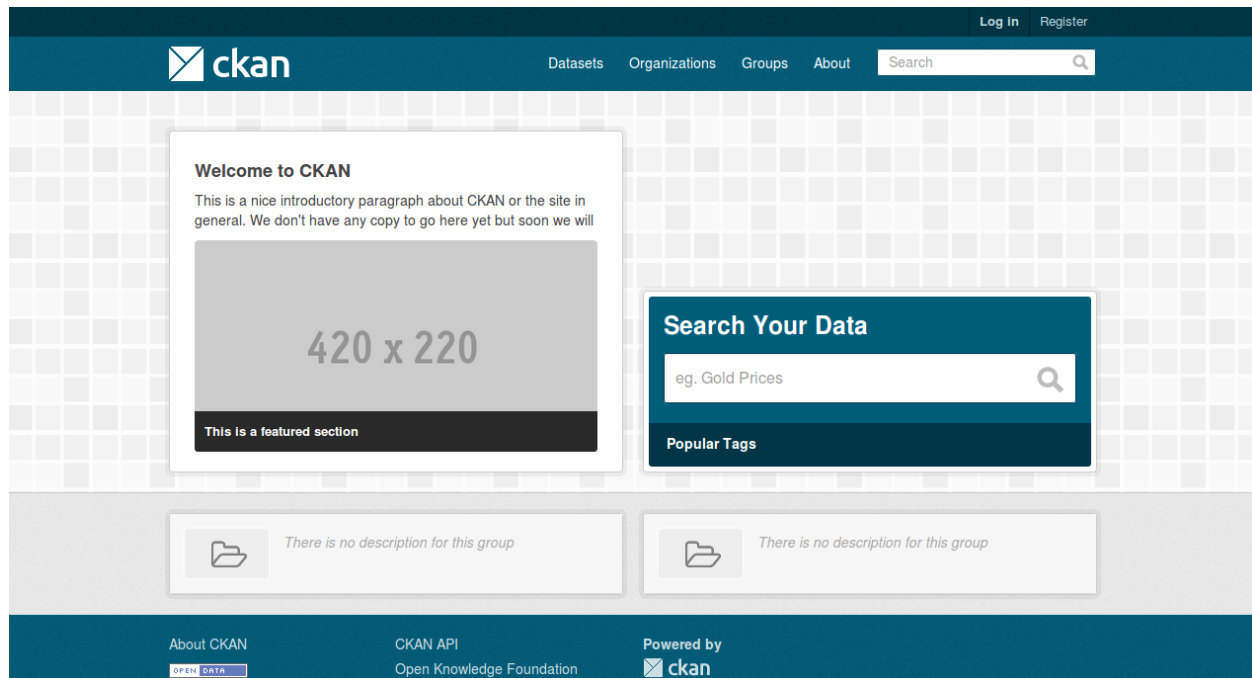
4. Restart Apache and Nginx

Restart Apache and Nginx by running this command in a terminal:

```
sudo service apache2 restart
sudo service nginx restart
```

5. You're done!

Open <http://localhost> in your web browser. You should see the CKAN front page, which will look something like this:



You can now move on to [Getting started](#) to begin using and customizing your CKAN site.

Note: The default authorization settings on a new install are deliberately restrictive. Regular users won't be able to create datasets or organizations. You should check the [Organizations and authorization](#) documentation, configure CKAN accordingly and grant other users the relevant permissions using the [sysadmin account](#).

Installing CKAN from source

This section describes how to install CKAN from source. Although [Installing CKAN from package](#) is simpler, it requires Ubuntu 12.04 64-bit. Installing CKAN from source works with other versions of Ubuntu and with other operating systems (e.g. RedHat, Fedora, CentOS, OS X). If you install CKAN from source on your own operating system, please share your experiences on our [How to Install CKAN](#) wiki page.

From source is also the right installation method for developers who want to work on CKAN.

1. Install the required packages

If you're using a Debian-based operating system (such as Ubuntu) install the required packages with this command:

```
sudo apt-get install python-dev postgresql libpq-dev python-pip python-virtualenv git-core solr-jetty
```

If you're not using a Debian-based operating system, find the best way to install the following packages on your operating system (see our [How to Install CKAN](#) wiki page for help):

Package	Description
Python	The Python programming language, v2.6 or 2.7
PostgreSQL	The PostgreSQL database system, v8.4 or newer
libpq	The C programmer's interface to PostgreSQL
pip	A tool for installing and managing Python packages
virtualenv	The virtual Python environment builder
Git	A distributed version control system
Apache Solr	A search platform
Jetty	An HTTP server (used for Solr).
OpenJDK 6 JDK	The Java Development Kit

2. Install CKAN into a Python virtual environment

Tip: If you're installing CKAN for development and want it to be installed in your home directory, you can symlink the directories used in this documentation to your home directory. This way, you can copy-paste the example commands from this documentation without having to modify them, and still have CKAN installed in your home directory:

```
mkdir -p ~/ckan/lib
sudo ln -s ~/ckan/lib /usr/lib/ckan
mkdir -p ~/ckan/etc
sudo ln -s ~/ckan/etc /etc/ckan
```

1. Create a Python [virtual environment](#) (virtualenv) to install CKAN into, and activate it:

```
sudo mkdir -p /usr/lib/ckan/default
sudo chown `whoami` /usr/lib/ckan/default
virtualenv --no-site-packages /usr/lib/ckan/default
. /usr/lib/ckan/default/bin/activate
```

Important: The final command above activates your virtualenv. The virtualenv has to remain active for the rest of the installation and deployment process, or commands will fail. You can tell when the virtualenv is active because its name appears in front of your shell prompt, something like this:

```
(default) $ _
```

For example, if you logout and login again, or if you close your terminal window and open it again, your virtualenv will no longer be activated. You can always reactivate the virtualenv with this command:

```
. /usr/lib/ckan/default/bin/activate
```

2. Install the CKAN source code into your virtualenv. To install the latest stable release of CKAN (CKAN 2.7.1), run:

```
pip install -e `git+https://github.com/ckan/ckan.git@ckan-2.7.1#egg=ckan`
```

If you're installing CKAN for development, you may want to install the latest development version (the most recent commit on the master branch of the CKAN git repository). In that case, run this command instead:

```
pip install -e `git+https://github.com/ckan/ckan.git#egg=ckan`
```

Warning: The development version may contain bugs and should not be used for production websites! Only install this version if you're doing CKAN development.

3. Install the Python modules that CKAN requires into your virtualenv:

Changed in version 2.1: In CKAN 2.0 and earlier the requirement file was called `pip-requirements.txt` not `requirements.txt` as below.

```
pip install -r /usr/lib/ckan/default/src/ckan/requirements.txt
```

4. Deactivate and reactivate your virtualenv, to make sure you're using the virtualenv's copies of commands like `paster` rather than any system-wide installed copies:

```
deactivate
. /usr/lib/ckan/default/bin/activate
```

3. Setup a PostgreSQL database

List existing databases:

```
sudo -u postgres psql -l
```

Check that the encoding of databases is UTF8, if not internationalisation may be a problem. Since changing the encoding of PostgreSQL may mean deleting existing databases, it is suggested that this is fixed before continuing with the CKAN install.

Next you'll need to create a database user if one doesn't already exist. Create a new PostgreSQL database user called `ckan_default`, and enter a password for the user when prompted. You'll need this password later:

```
sudo -u postgres createuser -S -D -R -P ckan_default
```

Create a new PostgreSQL database, called `ckan_default`, owned by the database user you just created:

```
sudo -u postgres createdb -O ckan_default ckan_default -E utf-8
```

Note: If PostgreSQL is run on a separate server, you will need to edit *postgresql.conf* and *pg_hba.conf*. For PostgreSQL 9.1 on Ubuntu, these files are located in *etc/postgresql/9.1/main*.

Uncomment the *listen_addresses* parameter and specify a comma-separated list of IP addresses of the network interfaces PostgreSQL should listen on or `*` to listen on all interfaces. For example,

```
listen_addresses = 'localhost,192.168.1.21'
```

Add a line similar to the line below to the bottom of *pg_hba.conf* to allow the machine running Apache to connect to PostgreSQL. Please change the IP address as desired according to your network settings.

```
host all all 192.168.1.22/32 md5
```

4. Create a CKAN config file

Create a directory to contain the site's config files:

```
sudo mkdir -p /etc/ckan/default
sudo chown -R `whoami` /etc/ckan/
```

Change to the `ckan` directory and create a CKAN config file:

```
cd /usr/lib/ckan/default/src/ckan
paster make-config ckan /etc/ckan/default/development.ini
```

Edit the `development.ini` file in a text editor, changing the following options:

sqlalchemy.url This should refer to the database we created in [3. Setup a PostgreSQL database](#) above:

```
sqlalchemy.url = postgresql://ckan_default:pass@localhost/ckan_default
```


Replace `pass` with the password that you created in 3. [Setup a PostgreSQL database](#) above.

Tip: If you're using a remote host with password authentication rather than SSL authentication, use:

```
sqlalchemy.url = postgresql://ckan_default:pass@<remotehost>/ckan_default?sslmode=disa
```

site_id Each CKAN site should have a unique `site_id`, for example:

```
ckan.site_id = default
```

site_url Provide the site's URL (used when putting links to the site into the FileStore, notification emails etc). For example:

```
ckan.site_url = http://demo.ckan.org
```

Do not add a trailing slash to the URL.

5. Setup Solr

CKAN uses [Solr](#) as its search platform, and uses a customized Solr schema file that takes into account CKAN's specific search needs. Now that we have CKAN installed, we need to install and configure Solr.

Note: These instructions explain how to setup Solr with a single core. If you want multiple applications, or multiple instances of CKAN, to share the same Solr server then you probably want a multi-core Solr setup instead. See [Multicore Solr setup](#).

Note: These instructions explain how to deploy Solr using the Jetty web server, but CKAN doesn't require Jetty - you can deploy Solr to another web server, such as Tomcat, if that's convenient on your operating system.

1. Edit the Jetty configuration file (`/etc/default/jetty`) and change the following variables:

```
NO_START=0           # (line 4)
JETTY_HOST=127.0.0.1 # (line 15)
JETTY_PORT=8983      # (line 18)
```

Start the Jetty server:

```
sudo service jetty start
```

You should now see a welcome page from Solr if you open <http://localhost:8983/solr/> in your web browser (replace localhost with your server address if needed).

Note: If you get the message Could not start Jetty servlet engine because no Java Development Kit (JDK) was found. then you will have to edit the `JAVA_HOME` setting in `/etc/default/jetty` to point to your machine's JDK install location. For example:

```
JAVA_HOME=/usr/lib/jvm/java-6-openjdk-amd64/
```

or:

```
JAVA_HOME=/usr/lib/jvm/java-6-openjdk-i386/
```

2. Replace the default `schema.xml` file with a symlink to the CKAN schema file included in the sources.

```
sudo mv /etc/solr/conf/schema.xml /etc/solr/conf/schema.xml.bak
sudo ln -s /usr/lib/ckan/default/src/ckan/ckan/config/solr/schema.xml /etc/solr/conf/s
```

Now restart Solr:

```
sudo service jetty restart
```

and check that Solr is running by opening <http://localhost:8983/solr/>.

3. Finally, change the `solr_url` setting in your CKAN config file to point to your Solr server, for example:

```
solr_url=http://127.0.0.1:8983/solr
```

6. Create database tables

Now that you have a configuration file that has the correct settings for your database, you can create the database tables:

```
cd /usr/lib/ckan/default/src/ckan
paster db init -c /etc/ckan/default/development.ini
```

You should see Initialising DB: SUCCESS.

Tip: If the command prompts for a password it is likely you haven't set up the `sqlalchemy.url` option in your CKAN configuration file properly. See 4. [Create a CKAN config file](#).

7. Set up the DataStore

Note: Setting up the DataStore is optional. However, if you do skip this step, the *DataStore features* will not be available and the DataStore tests will fail.

Follow the instructions in *DataStore extension* to create the required databases and users, set the right permissions and set the appropriate values in your CKAN config file.

8. Link to who.ini

`who.ini` (the Repoze.who configuration file) needs to be accessible in the same directory as your CKAN config file, so create a symlink to it:

```
ln -s /usr/lib/ckan/default/src/ckan/who.ini /etc/ckan/default/who.ini
```

9. You're done!

You can now use the Paste development server to serve CKAN from the command-line. This is a simple and lightweight way to serve CKAN that is useful for development and testing:

```
cd /usr/lib/ckan/default/src/ckan
paster serve /etc/ckan/default/development.ini
```

Open <http://127.0.0.1:5000/> in a web browser, and you should see the CKAN front page.

Now that you've installed CKAN, you should:

- Run CKAN's tests to make sure that everything's working, see *Testing CKAN*.
- If you want to use your CKAN site as a production site, not just for testing or development purposes, then deploy CKAN using a production web server such as Apache or Nginx. See *Deploying a source install*.
- Begin using and customizing your site, see *Getting started*.

Note: The default authorization settings on a new install are deliberately restrictive. Regular users won't be able to create datasets or organizations. You should check the [Organizations and authorization](#) documentation, configure CKAN accordingly and grant other users the relevant permissions using the [sysadmin account](#).

Source install troubleshooting

Solr setup troubleshooting

Solr requests and errors are logged in the web server log files.

- For Jetty servers, the log files are:

```
/var/log/jetty/<date>.stderrout.log
```

- For Tomcat servers, they're:

```
/var/log/tomcat6/catalina.<date>.log
```

Unable to find a javac compiler If when running Solr it says:

Unable to find a javac compiler; com.sun.tools.javac.Main is not on the classpath. Perhaps JAVA_HOME does not point to the JDK.

See the note in [5. Setup Solr](#) about `JAVA_HOME`. Alternatively you may not have installed the JDK. Check by seeing if `javac` is installed:

```
which javac
```

If `javac` isn't installed, do:

```
sudo apt-get install openjdk-6-jdk
```

and then restart Solr:

```
sudo service jetty restart
```

AttributeError: 'module' object has no attribute 'css/main.debug.css' This error is likely to show up when `debug` is set to `True`. To fix this error, install frontend dependencies. See [Frontend development guidelines](#).

After installing the dependencies, run `bin/less` and then start paster server again.

If you do not want to compile CSS, you can also copy the `main.css` to `main.debug.css` to get CKAN running.

```
cp /usr/lib/ckan/default/src/ckan/ckan/public/base/css/main.css /usr/lib/ckan/default/src/ckan/ckan/public/base/css/main.debug.css
```

Installing CKAN using a Docker image

Important: These instructions require Docker `>=1.0`. The released version of Docker is 1.0.1 as at this writing.

Note: Installing CKAN using a Docker image is currently under evaluation. There may be omissions or inaccuracies in this documentation. In particular, the current Docker image omits the configuration required to run the `DataStore/DataPusher`. Proceed with caution.

This section describes how to install CKAN using a [Docker](#) image. Docker is a tool that allows all kinds of software to be shipped and deployed in a standard format, much as cargo is shipped around the world in [standardised shipping containers](#).

CKAN is built into a binary image, which you can then use as a blueprint to launch multiple containers which run CKAN and attendant services in an isolated environment. Providing a full introduction to Docker concepts is out of the scope of this document: you can learn more in the [Docker documentation](#).

Prerequisites

In order to install CKAN using Docker, you will need to have installed Docker. Please follow the instructions for installing Docker from [the Docker documentation](#).

Installing CKAN

In the simplest case, installing CKAN should be a matter of running three commands: to run PostgreSQL, Solr, and CKAN:

```
$ docker run -d --name db ckan/postgresql
$ docker run -d --name solr ckan/solr
$ docker run -d -p 80:80 --link db:db --link solr:solr ckan/ckan
```

This starts a new CKAN container in the background, connected to default installations of PostgreSQL and Solr also running in containers.

Warning: The default PostgreSQL container is INAPPROPRIATE FOR PRODUCTION USE. The default database username and password is “ckan:ckan” and if you do not change this the contents of your database may well be exposed to the public.

Note: The first time you run these `docker run` commands, Docker will have to download the software images: this may be quite slow. Once you’ve downloaded the images, however, subsequent calls to `docker run` will be much faster. If you want, you can run `echo postgresql solr ckan | xargs -n1 -IIMG docker pull ckan/IMG` to pre-fetch the images.

If all goes well you should now have a CKAN instance running. You can use `docker ps` to verify that your container started. You should see something like the following:

```
$ docker ps
CONTAINER ID      IMAGE               COMMAND             CREATED           STATUS
cab6e63c77b1     ckan/ckan:latest   /sbin/my_init       30 days ago      Up 1 minute
fb47b3744d6d     ckan/postgresql:latest   /usr/local/bin/run  9 days ago       Up 1 minute
96e963812fc9     ckan/solr:latest   java -jar start.jar  15 days ago      Up 1 minute
```

Using the CKAN container id (here it’s `cab6e63c77b1`), you can perform other actions on your container, such as viewing the logs:

```
$ docker logs cab6e63c77b1
```

or stopping the container:

```
$ docker stop cab6e63c77b1
```

If you wish to run CKAN on a different port or bind it to a specific IP address on the machine, please consult the output of `docker help run` to see valid values for the `-p/--publish` option.

You can also configure the CKAN container to connect to remote PostgreSQL and Solr services, without using Docker links, by setting the `DATABASE_URL` and `SOLR_URL` environment variables:

```
$ docker run -d -p 80:80 \
  -e DATABASE_URL=postgresql://ckanuser:password@192.168.0.5/ckan \
  -e SOLR_URL=http://192.168.0.6:8983/solr/ckan
```

Running maintenance commands

Note: This is currently more fiddly than we would like, and we will hopefully soon add a helper command to make this easier.

You can run maintenance commands in their own ephemeral container by specifying a custom command for the container. For example, to create a sysadmin user called `joebloggs`:

```
$ docker run -i -t --link db:db --link solr:solr \
  ckan/ckan \
  /sbin/my_init -- \
  /bin/bash -c \
  '$CKAN_HOME/bin/paster --plugin=ckan sysadmin -c $CKAN_CONFIG/ckan.ini add joebloggs'
```

Customizing the Docker image

You may well find you want to customize your CKAN installation, either by setting custom configuration options not exposed by the Docker image, or by installing additional CKAN extensions. A full guide to extending Docker images is out-of-scope of this installation documentation, but you can use the functionality provided by `docker build` to extend the `ckan/ckan` image: <http://docs.docker.com/reference/builder/>.

For example, if you wanted custom configuration and the CKAN Spatial extension, you could build an image from a Dockerfile like the following:

```
FROM ckan/ckan

# Install git
RUN DEBIAN_FRONTEND=noninteractive apt-get update
RUN DEBIAN_FRONTEND=noninteractive apt-get install -q -y git

# Install the CKAN Spatial extension
RUN $CKAN_HOME/bin/pip install -e git+https://github.com/ckan/ckanext-spatial.git@stable#egg=ckanext-

# Add my custom configuration file
ADD mycustomconfig.ini $CKAN_CONFIG/ckan.ini
```

You would then reference your built image instead of `ckan/ckan` when calling the `docker run` commands listed above.

Deploying a source install

Once you've installed CKAN from source by following the instructions in *Installing CKAN from source*, you can follow these instructions to deploy your CKAN site using a production web server (Apache), so that it's available to the Internet.

Note: If you installed CKAN from package you don't need to follow this section, your site is already deployed using Apache and modwsgi as described below.

Because CKAN uses WSGI, a standard interface between web servers and Python web applications, CKAN can be used with a number of different web server and deployment configurations including:

- [Apache](#) with the modwsgi Apache module proxied with [Nginx](#) for caching
- [Apache](#) with the modwsgi Apache module
- [Apache](#) with paster and reverse proxy
- [Nginx](#) with paster and reverse proxy
- [Nginx](#) with uwsgi

This guide explains how to deploy CKAN using Apache and modwsgi and proxied with Nginx on an Ubuntu server. These instructions have been tested on Ubuntu 12.04.

If run into any problems following these instructions, see [Troubleshooting](#) below.

1. Create a `production.ini` File

Create your site's `production.ini` file, by copying the `development.ini` file you created in [Installing CKAN from source](#) earlier:

```
cp /etc/ckan/default/development.ini /etc/ckan/default/production.ini
```

2. Install Apache, modwsgi, modrpaf

Install [Apache](#) (a web server), [modwsgi](#) (an Apache module that adds WSGI support to Apache), and [modrpaf](#) (an Apache module that sets the right IP address when there is a proxy forwarding to Apache):

```
sudo apt-get install apache2 libapache2-mod-wsgi libapache2-mod-rpaf
```

3. Install Nginx

Install [Nginx](#) (a web server) which will proxy the content from [Apache](#) and add a layer of caching:

```
sudo apt-get install nginx
```

4. Install an email server

If one isn't installed already, install an email server to enable CKAN's email features (such as sending traceback emails to sysadmins when crashes occur, or sending new activity [email notifications](#) to users). For example, to install the [Postfix](#) email server, do:

```
sudo apt-get install postfix
```

When asked to choose a Postfix configuration, choose *Internet Site* and press return.

5. Create the WSGI script file

Create your site's WSGI script file `/etc/ckan/default/apache.wsgi` with the following contents:

```
import os
activate_this = os.path.join('/usr/lib/ckan/default/bin/activate_this.py')
execfile(activate_this, dict(__file__=activate_this))

from paste.deploy import loadapp
config_filepath = os.path.join(os.path.dirname(os.path.abspath(__file__)), 'production.ini')
from paste.script.util.logging_config import fileConfig
fileConfig(config_filepath)
application = loadapp('config:%s' % config_filepath)
```

The modwsgi Apache module will redirect requests to your web server to this WSGI script file. The script file then handles those requests by directing them on to your CKAN instance (after first configuring the Python environment for CKAN to run in).

6. Create the Apache config file

Create your site's Apache config file at `/etc/apache2/sites-available/ckan_default`, with the following contents:

```
<VirtualHost 127.0.0.1:8080>
    ServerName default.ckanhosted.com
    ServerAlias www.default.ckanhosted.com
    WSGIScriptAlias / /etc/ckan/default/apache.wsgi

    # Pass authorization info on (needed for rest api).
    WSGIPassAuthorization On

    # Deploy as a daemon (avoids conflicts between CKAN instances).
    WSGIDaemonProcess ckan_default display-name=ckan_default processes=2 threads=15

    WSGIProcessGroup ckan_default

    ErrorLog /var/log/apache2/ckan_default.error.log
    CustomLog /var/log/apache2/ckan_default.custom.log combined

    <IfModule mod_rpaf.c>
        RPAFenable On
        RPAFsethostname On
        RPAFproxy_ips 127.0.0.1
    </IfModule>
</VirtualHost>
```

Replace `default.ckanhosted.com` and `www.default.ckanhosted.com` with the domain name for your site.

This tells the Apache modwsgi module to redirect any requests to the web server to the WSGI script that you created above. Your WSGI script in turn directs the requests to your CKAN instance.

7. Modify the Apache ports.conf file

Open `/etc/apache2/ports.conf`. Look in the file for the following lines:

```
NameVirtualHost *:80
Listen 80
```

Change the entries from 80 to 8080 to look like the following:

```
NameVirtualHost *:8080
Listen 8080
```

8. Create the Nginx config file

Create your site's Nginx config file at `/etc/nginx/sites-available/ckan_default`, with the following contents:

```
proxy_cache_path /tmp/nginx_cache levels=1:2 keys_zone=cache:30m max_size=250m;
proxy_temp_path /tmp/nginx_proxy 1 2;

server {
    client_max_body_size 100M;
    location / {
        proxy_pass http://127.0.0.1:8080/;
        proxy_set_header X-Forwarded-For $remote_addr;
        proxy_set_header Host $host;
        proxy_cache cache;
        proxy_cache_bypass $cookie_auth_tkt;
        proxy_no_cache $cookie_auth_tkt;
        proxy_cache_valid 30m;
        proxy_cache_key $host$scheme$proxy_host$request_uri;
        # In emergency comment out line to force caching
        # proxy_ignore_headers X-Accel-Expires Expires Cache-Control;
    }
}
```

9. Enable your CKAN site

To prevent conflicts, disable your default nginx and apache sites. Finally, enable your CKAN site in Apache:

```
sudo a2ensite ckan_default
sudo a2dissite default
sudo rm -vi /etc/nginx/sites-enabled/default
sudo ln -s /etc/nginx/sites-available/ckan_default /etc/nginx/sites-enabled/ckan_default
sudo service apache2 reload
sudo service nginx reload
```

You should now be able to visit your server in a web browser and see your new CKAN instance.

Troubleshooting

Default Apache welcome page

If you see a default Apache welcome page where your CKAN front page should be, it may be because the default Apache config file is overriding your CKAN config file (both use port 80), so disable it and restart Apache:

```
sudo a2dissite default
sudo service apache2 reload
```

403 Forbidden and 500 Internal Server Error

If you see a 403 Forbidden or 500 Internal Server Error page where your CKAN front page should be, you may have a problem with your unix file permissions. The Apache web server needs to have permission to access your WSGI script

file and all of its parent directories. The permissions of the file should look like `-rw-r--r--` and the permissions of each of its parent directories should look like `drwxr-xr-x`.

IOError: sys.stdout access restricted by mod_wsgi

If you're getting 500 Internal Server Error pages and you see `IOError: sys.stdout access restricted by mod_wsgi` in your log files, it means that something in your WSGI application (e.g. your WSGI script file, your CKAN instance, or one of your CKAN extensions) is trying to print to stdout, for example by using standard Python print statements. WSGI applications are not allowed to write to stdout. Possible solutions include:

1. Remove the offending print statements. One option is to replace print statements with statements like `print >> sys.stderr, "..."`
2. Redirect all print statements to stderr:

```
import sys
sys.stdout = sys.stderr
```
3. Allow your application to print to stdout by putting `WSGIRestrictStdout Off` in your Apache config file (not recommended).

Also see <https://code.google.com/p/modwsgi/wiki/ApplicationIssues>

Log files

In general, if it's not working look in the log files in `/var/log/apache2` for error messages. `ckan_default.error.log` should be particularly interesting.

modwsgi wiki

Some pages on the modwsgi wiki have some useful information for troubleshooting modwsgi problems:

- <https://code.google.com/p/modwsgi/wiki/ApplicationIssues>
- <http://code.google.com/p/modwsgi/wiki/DebuggingTechniques>
- <http://code.google.com/p/modwsgi/wiki/QuickConfigurationGuide>
- <http://code.google.com/p/modwsgi/wiki/ConfigurationGuidelines>
- <http://code.google.com/p/modwsgi/wiki/FrequentlyAskedQuestions>
- <http://code.google.com/p/modwsgi/wiki/ConfigurationIssues>

Upgrading CKAN

This document describes the different types of CKAN release, and explains how to upgrade a site to a newer version of CKAN.

See also:

Changelog The changelog lists all CKAN releases and the main changes introduced in each release.

Doing a CKAN release Documentation of the process that the CKAN developers follow to do a CKAN release.

CKAN releases

CKAN follows a predictable release cycle so that users can depend on stable releases of CKAN, and can plan their upgrades to new releases.

Each release has a version number of the form $M.m$ (eg. 2.1) or $M.m.p$ (eg. 1.8.2), where M is the **major version**, m is the **minor version** and p is the **patch version** number. There are three types of release:

Major Releases Major releases, such as CKAN 1.0 and CKAN 2.0, increment the major version number. These releases contain major changes in the CKAN code base, with significant refactorings and breaking changes, for instance in the API or the templates. These releases are very infrequent.

Minor Releases Minor releases, such as CKAN 1.8 and CKAN 2.1, increment the minor version number. These releases are not as disruptive as major releases, but backwards-incompatible changes *may* be introduced in minor releases. The [Changelog](#) will document any breaking changes. We aim to release a minor version of CKAN roughly every three months.

Patch Releases Patch releases, such as CKAN 1.8.1 or CKAN 2.0.1, increment the patch version number. These releases do not break backwards-compatibility, they include only bug fixes and non-breaking optimizations and features. Patch releases do not contain:

- Database schema changes or migrations
- Function interface changes
- Plugin interface changes
- New dependencies
- Big refactorings or new features in critical parts of the code

Note: Users should always run the latest patch release for the minor release they are on, as patch releases contain important bug fixes and security updates. Because patch releases don't include backwards incompatible changes, the upgrade process (as described in [Upgrading a CKAN 2 package install to a new patch release](#)) should be straightforward.

Outdated patch releases will no longer be supported after a newer patch release has been released. For example once CKAN 2.0.2 has been released, CKAN 2.0.1 will no longer be supported.

Releases are announced on the [ckan-announce mailing list](#), a low-volume list that CKAN instance maintainers can subscribe to in order to be up to date with upcoming releases.

Upgrading CKAN

This section will walk you through the steps to upgrade your CKAN site to a newer version of CKAN.

Note: Before upgrading your version of CKAN you should check that any custom templates or extensions you're using work with the new version of CKAN. For example, you could install the new version of CKAN in a new virtual environment and use that to test your templates and extensions.

Note: You should also read the [Changelog](#) to see if there are any extra notes to be aware of when upgrading to the new version.

1. Backup your database

You should always backup your CKAN database before upgrading CKAN. If something goes wrong with the CKAN upgrade you can use the backup to restore the database to its pre-upgrade state.

1. Activate your virtualenv and switch to the ckan source directory, e.g.:

```
. /usr/lib/ckan/default/bin/activate
cd /usr/lib/ckan/default/src/ckan
```

2. Backup your CKAN database using the `db dump` command, for example:

```
paster db dump --config=/etc/ckan/default/development.ini my_ckan_database.pg_dump
```

This will create a file called `my_ckan_database.pg_dump`, you can use the `db load` command to restore your database to the state recorded in this file. See [db: Manage databases](#) for details of the `db dump` and `db load` commands.

2. Upgrade CKAN

The process of upgrading CKAN differs depending on whether you have a package install or a source install of CKAN, and whether you're upgrading to a *major*, *minor* or *patch release* of CKAN. Follow the appropriate one of these documents:

Upgrading a CKAN 1 package install to CKAN 2.0

Note: If you want to upgrade a CKAN 1.x package install to a newer version of CKAN 1 (as opposed to upgrading to CKAN 2), see the [documentation](#) relevant to the old CKAN packaging system.

The CKAN 2.0 package requires Ubuntu 12.04 64-bit, whereas previous CKAN packages used Ubuntu 10.04. CKAN 2.0 also introduces many backwards-incompatible feature changes (see [the changelog](#)). So it's not possible to automatically upgrade to a CKAN 2.0 package install.

However, you can install CKAN 2.0 (either on the same server that contained your CKAN 1.x site, or on a different machine) and then manually migrate your database and any custom configuration, extensions or templates to your new CKAN 2.0 site. We will outline the main steps for migrating below.

1. Create a dump of your CKAN 1.x database:

```
sudo -u ckanstd /var/lib/ckan/std/pyenv/bin/paster --plugin=ckan db dump db-1.x.dump --config=/e
```

2. If you want to install CKAN 2.0 on the same server that your CKAN 1.x site was on, uninstall the CKAN 1.x package first:

```
sudo apt-get autoremove ckan
```

3. Install CKAN 2.0, either from a [package install](#) if you have Ubuntu 12.04 64-bit, or from a [source install](#) otherwise.
4. Load your database dump from CKAN 1.x into CKAN 2.0. This will migrate all of your datasets, resources, groups, tags, user accounts, and other data to CKAN 2.0. Your database schema will be automatically upgraded, and your search index rebuilt.

First, activate your CKAN virtual environment and change to the ckan dir:

```
. /usr/lib/ckan/default/bin/activate
cd /usr/lib/ckan/default/src/ckan
```

Now, load your database. **This will delete any data already present in your new CKAN 2.0 database.** If you've installed CKAN 2.0 on a different machine from 1.x, first copy the database dump file to that machine. Then run these commands:

```
paster db clean -c /etc/ckan/default/production.ini
paster db load -c /etc/ckan/default/production.ini db-1.x.dump
```

5. If you had any custom config settings in your CKAN 1.x instance that you want to copy across to your CKAN 2.0 instance, then update your CKAN 2.0 `/etc/ckan/default/production.ini` file with these config settings. Note that not all CKAN 1.x config settings are still supported in CKAN 2.0, see [Configuration Options](#) for details.

In particular, CKAN 2.0 introduces an entirely new authorization system and any custom authorization settings you had in CKAN 1.x will have to be reconsidered for CKAN 2.0. See [Organizations and authorization](#) for details.

6. If you had any extensions installed in your CKAN 1.x instance that you also want to use with your CKAN 2.0 instance, install those extensions in CKAN 2.0. Not all CKAN 1.x extensions are compatible with CKAN 2.0. Check each extension's documentation for CKAN 2.0 compatibility and install instructions.
7. If you had any custom templates in your CKAN 1.x instance, these will need to be adapted before they can be used with CKAN 2.0. CKAN 2.0 introduces an entirely new template system based on Jinja2 rather than on Genshi. See [Theming guide](#) for details.

Upgrading a CKAN 2 package install to a new patch release

Note: Before upgrading CKAN you should check the compatibility of any custom themes or extensions you're using, check the changelog, and backup your database. See [Upgrading CKAN](#).

[Patch releases](#) are distributed in the same package as the minor release they belong to, so for example CKAN 2.0, 2.0.1, 2.0.2, etc. will all be installed using the CKAN 2.0 package (`python-ckan_2.0_amd64.deb`):

1. Download the CKAN package:

```
wget http://packaging.ckan.org/python-ckan_2.0_amd64.deb
```

You can check the actual CKAN version from a package running the following command:

```
dpkg --info python-ckan_2.0_amd64.deb
```

Look for the `Version` field in the output:

```
...
Package: python-ckan
Version: 2.0.1-3
...
```

2. Install the package with the following command:

```
sudo dpkg -i python-ckan_2.0_amd64.deb
```

Your CKAN instance should be upgraded straight away.

Note: If you have changed the Apache, Nginx or `who.ini` configuration files, you will get a prompt like the following, asking whether to keep your local changes or replace the files. You generally would like to keep your local changes (option `N`, which is the default), but you can look at the differences between versions by selecting option `D`:

```
Configuration file `/etc/apache2/sites-available/ckan_default'
==> File on system created by you or by a script.
==> File also in package provided by package maintainer.
What would you like to do about it ? Your options are:
  Y or I : install the package maintainer's version
  N or O : keep your currently-installed version
  D      : show the differences between the versions
  Z      : start a shell to examine the situation
The default action is to keep your current version.
*** ckan_default (Y/I/N/O/D/Z) [default=N] ?
```

Your local CKAN configuration file in `/etc/ckan/default` will not be replaced.

Note: The install process will uninstall any existing CKAN extensions or other libraries located in the `src` directory of the CKAN virtualenv. To enable them again, the installation process will iterate all folders in the `src` directory, reinstall the requirements listed in `pip-requirements.txt` and `requirements.txt` files and run `python setup.py develop` for each. If you are using a custom extension which does not use this requirements file names or is located elsewhere, you will need to manually reenale it.

Note: When upgrading from 2.0 to 2.0.1 you may see some vdm related warnings when installing the package:

```
dpkg: warning: unable to delete old directory '/usr/lib/ckan/default/src/vdm': Directory not empty
```

These are due to vdm not longer being installed from source. You can ignore them and delete the folder manually if you want.

3. Finally, restart Apache and Nginx:

```
sudo service apache2 restart
sudo service nginx restart
```

4. You're done!

You should now be able to visit your CKAN website in your web browser and see that it's running the new version of CKAN.

Upgrading a CKAN 2 package install to a new minor release

Note: Before upgrading CKAN you should check the compatibility of any custom themes or extensions you're using, check the changelog, and backup your database. See [Upgrading CKAN](#).

Each *minor release* is distributed in its own package, so for example CKAN 2.0.X and 2.1.X will be installed using the `python-ckan_2.0_amd64.deb` and `python-ckan_2.1_amd64.deb` packages respectively.

1. Download the CKAN package for the new minor release you want to upgrade to (replace the version number with the relevant one):

```
wget http://packaging.ckan.org/python-ckan_2.1_amd64.deb
```

2. Install the package with the following command:

```
sudo dpkg -i python-ckan_2.1_amd64.deb
```

Note: If you have changed the Apache, Nginx or `who.ini` configuration files, you will get a prompt like the following, asking whether to keep your local changes or replace the files. You generally would like to keep your

local changes (option N, which is the default), but you can look at the differences between versions by selecting option D:

```
Configuration file `/etc/apache2/sites-available/ckan_default'
==> File on system created by you or by a script.
==> File also in package provided by package maintainer.
What would you like to do about it ? Your options are:
  Y or I : install the package maintainer's version
  N or O : keep your currently-installed version
  D      : show the differences between the versions
  Z      : start a shell to examine the situation
The default action is to keep your current version.
*** ckan_default (Y/I/N/O/D/Z) [default=N] ?
```

Your local CKAN configuration file in `/etc/ckan/default` will not be replaced.

Note: The install process will uninstall any existing CKAN extensions or other libraries located in the `src` directory of the CKAN virtualenv. To enable them again, the installation process will iterate over all folders in the `src` directory, reinstall the requirements listed in `pip-requirements.txt` and `requirements.txt` files and run `python setup.py develop` for each. If you are using a custom extension which does not use this requirements file name or is located elsewhere, you will need to manually reinstall it.

3. If there have been changes in the database schema (check the [Changelog](#) to find out) you need to update your CKAN database's schema using the `db upgrade` command.

Warning: To avoid problems during the database upgrade, comment out any plugins that you have enabled in your ini file. You can uncomment them again when the upgrade finishes.

For example:

```
paster db upgrade --config=/etc/ckan/default/development.ini
```

See [db: Manage databases](#) for details of the `db upgrade` command.

4. If there have been changes in the Solr schema (check the [Changelog](#) to find out) you need to restart Jetty for the changes to take effect:

```
sudo service jetty restart
```

5. If you have any CKAN extensions installed from source, you may need to checkout newer versions of the extensions that work with the new CKAN version. Refer to the documentation for each extension. We recommend disabling all extensions on your ini file and re-enable them one by one to make sure they are working fine.

6. Rebuild your search index by running the `ckan search-index rebuild` command:

```
sudo ckan search-index rebuild -r
```

See [search-index: Rebuild search index](#) for details of the `ckan search-index rebuild` command.

7. Finally, restart Apache and Nginx:

```
sudo service apache2 restart
sudo service nginx restart
```

Upgrading a source install

Note: Before upgrading CKAN you should check the compatibility of any custom themes or extensions you're using,

check the changelog, and backup your database. See [Upgrading CKAN](#).

The process for upgrading a source install is the same, no matter what type of CKAN release you're upgrading to:

1. Activate your virtualenv and switch to the ckan source directory, e.g.:

```
. /usr/lib/ckan/default/bin/activate
cd /usr/lib/ckan/default/src/ckan
```

2. Checkout the new CKAN version from git, for example:

```
git fetch
git checkout release-v2.0
```

If you have any CKAN extensions installed from source, you may need to checkout newer versions of the extensions at this point as well. Refer to the documentation for each extension.

3. Update CKAN's dependencies:

Changed in version 2.1: In CKAN 2.0 and earlier the requirements file was called `pip-requirements.txt`, not `requirements.txt` as below.

```
pip install --upgrade -r requirements.txt
```

4. Register any new or updated plugins:

```
python setup.py develop
```

5. If there have been changes in the Solr schema (check the [Changelog](#) to find out) you need to restart Jetty for the changes to take effect:

```
sudo service jetty restart
```

6. If you are upgrading to a new [major release](#) update your CKAN database's schema using the `db upgrade` command.

Warning: To avoid problems during the database upgrade, comment out any plugins that you have enabled in your ini file. You can uncomment them again when the upgrade finishes.

For example:

```
paster db upgrade --config=/etc/ckan/default/development.ini
```

See [db: Manage databases](#) for details of the `db upgrade` command.

7. Rebuild your search index by running the `ckan search-index rebuild` command:

```
paster search-index rebuild -r --config=/etc/ckan/default/development.ini
```

See [search-index: Rebuild search index](#) for details of the `ckan search-index rebuild` command.

8. Finally, restart your web server. For example if you have deployed CKAN using the Apache web server on Ubuntu linux, run this command:

```
sudo service apache2 reload
```

9. You're done!

You should now be able to visit your CKAN website in your web browser and see that it's running the new version of CKAN.

Getting started

Once you've finished *installing CKAN*, this section will walk you through getting started with your new CKAN website, including creating a CKAN sysadmin user, some test data, and the basics of configuring your CKAN site.

Creating a sysadmin user

You have to use CKAN's command line interface to create your first sysadmin user, and it can also be useful to create some test data from the command line. For full documentation of CKAN's command line interface (including troubleshooting) see *Command Line Interface*.

Note: CKAN commands are executed using the `paster` command on the server that CKAN is installed on. Before running the paster commands below, you need to make sure that your virtualenv is activated and that you're in your ckan source directory. For example:

```
. /usr/lib/ckan/default/bin/activate
cd /usr/lib/ckan/default/src/ckan
```

You have to create your first CKAN sysadmin user from the command line. For example, to create a user called `seanh` and make him a sysadmin:

```
paster sysadmin add seanh -c /etc/ckan/default/production.ini
```

If a user called `seanh` already exists he will be promoted to a sysadmin. If the user account doesn't exist yet you'll be prompted to enter a password and the account will be created.

For a list of other command line commands for managing sysadmins, run:

```
paster sysadmin --help
```

Read the *Sysadmin guide* to learn what you can do as a CKAN sysadmin.

Creating test data

It can be handy to have some test data to start with, to quickly check that everything works. You can add a standard set of test data to your site from the command line with the `create-test-data` command:

```
paster create-test-data -c /etc/ckan/default/production.ini
```

If you later want to delete this test data and start again with an empty database, you can use the `db clean` command, see *db: Manage databases*.

For a list of other command line commands for creating tests data, run:

```
paster create-test-data --help
```

Config file

All of the options that can be set in the admin page and many more can be set by editing CKAN's config file. By default, the config file is located at `/etc/ckan/default/development.ini` for development sites or `/etc/ckan/default/production.ini` for production sites. The config file can be edited in any text editor. For example, to change the title of your site you would find the `ckan.site_title` line in your config file and edit it:

```
ckan.site_title = Masag Data Hub
```


Make sure the line is not commented-out (lines in the config file that begin with # are considered comments, so if there's a # at the start of a line you've edited, delete it), save the file, and then restart your web server for the changes to take effect. For example, if using Apache on Ubuntu:

```
sudo service apache2 reload
```

For full documentation of CKAN's config file and all the options you can set, see [Configuration Options](#).

Note: If the same option is set in both the config file and in the admin page, the admin page setting takes precedence. You can use the *Reset* button on the admin page to clear your settings, and allow settings from the config file to take effect.

Command Line Interface

Most common CKAN administration tasks can be carried out from the command line on the server that CKAN is installed on, using the `paster` command.

If you have trouble running paster commands, see [Troubleshooting Paster Commands](#) below.

Note: Before running a CKAN `paster` command, you have to activate your CKAN virtualenv and change to the `ckan` directory, for example:

```
. /usr/lib/ckan/default/bin/activate
cd /usr/lib/ckan/default/src/ckan
```

To run a paster command without activating the virtualenv first, you have to give the full path the paster script within the virtualenv, for example:

```
/usr/lib/ckan/default/bin/paster --plugin=ckan user list -c /etc/ckan/default/development.ini
```

To run a paster command without changing to the `ckan` directory first, add the `--plugin=ckan` option to the command. For example:

```
paster --plugin=ckan user list -c /etc/ckan/default/development.ini
```

In the example commands below, we assume you're running the commands with your virtualenv activated and from your `ckan` directory.

The general form of a CKAN `paster` command is:

```
paster command --config=/etc/ckan/default/development.ini
```

The `--config` option tells CKAN where to find your config file, which it reads for example to know which database it should use. As you'll see in the examples below, this option can be given as `-c` for short.

`command` should be replaced with the name of the CKAN command that you wish to execute. Most commands have their own subcommands and options. For example, to print out a list of all of your CKAN site's users do:

Note: You may also specify the location of your config file using the `CKAN_INI` environment variable. You will no longer need to use `--config=` or `-c=` to tell paster where the config file is:

```
export CKAN_INI=/etc/ckan/default/development.ini
```

```
paster user list -c /etc/ckan/default/development.ini
```

(Here `user` is the name of the CKAN command you're running, and `list` is a subcommand of `user`.)

For a list of all available commands, simply run `paster` on its own with no command, or see [Paster Commands Reference](#). In this case we don't need the `-c` option, since we're only asking CKAN to print out information about commands, not to actually do anything with our CKAN site:

```
paster
```

Each command has its own help text, which tells you what subcommands and options it has (if any). To print out a command's help text, run the command with the `--help` option:

```
paster user --help
```

Troubleshooting Paster Commands

Permission Error

If you receive 'Permission Denied' error, try running `paster` with `sudo`.

```
sudo /usr/lib/ckan/default/bin/paster db clean -c /etc/ckan/default/production.ini
```

Virtualenv not activated, or not in ckan dir

Most errors with `paster` commands can be solved by remembering to **activate your virtual environment** and **change to the ckan directory** before running the command:

```
. /usr/lib/ckan/default/bin/activate
cd /usr/lib/ckan/default/src/ckan
```

Error messages such as the following are usually caused by forgetting to do this:

- **Command 'foo' not known** (where *foo* is the name of the command you tried to run)
- **The program 'paster' is currently not installed**
- **Command not found: paster**
- **ImportError: No module named fanstatic** (or other `ImportErrors`)

Running paster commands provided by extensions

If you're trying to run a CKAN command provided by an extension that you've installed and you're getting an error like **Command 'foo' not known** even though you've activated your virtualenv and changed to the ckan directory, this is because you need to run the extension's `paster` commands from the extension's source directory not CKAN's source directory. For example:

```
. /usr/lib/ckan/default/bin/activate
cd /usr/lib/ckan/default/src/ckanext-spatial
paster foo -c /etc/ckan/default/development.ini
```

This should not be necessary when using the pre-installed extensions that come with CKAN.

Alternatively, you can give the extension's name using the `--plugin` option, for example

```
paster --plugin=ckanext-foo foo -c /etc/ckan/default/development.ini
```

Todo

Running a `paster` shell with `paster --plugin=pylons shell -c` Useful for development?

Wrong config file path

AssertionError: Config filename development.ini does not exist This means you forgot to give the `--config` or `-c` option to tell CKAN where to find your config file. (CKAN looks for a config file named `development.ini` in your current working directory by default.)

ConfigParser.MissingSectionHeaderError: File contains no section headers This happens if the config file that you gave with the `-c` or `--config` option is badly formatted, or if you gave the wrong filename.

IOError: [Errno 2] No such file or directory: ‘...’ This means you gave the wrong path to the `--config` or `-c` option (you gave a path to a file that doesn’t exist).

Paster Commands Reference

The following paster commands are supported by CKAN:

celeryd	Control celery daemon.
check-po-files	Check po files for common mistakes
color	Create or remove a color scheme.
create-test-data	Create test data in the database.
dataset	Manage datasets.
datastore	Perform commands to set up the datastore.
db	Perform various tasks on the database.
front-end-build	Creates and minifies css and JavaScript files
less	Compile all root less documents into their CSS counterparts
minify	Create minified versions of the given Javascript and CSS files.
notify	Send out modification notifications.
plugin-info	Provide info on installed plugins.
profile	Code speed profiler
ratings	Manage the ratings stored in the db
rdf-export	Export active datasets as RDF.
search-index	Creates a search index for all datasets
sysadmin	Gives sysadmin rights to a named user.
tracking	Update tracking statistics.
trans	Translation helper functions
user	Manage users.

celeryd: Control celery daemon

Usage:

```
celeryd <run>           - run the celery daemon
celeryd run concurrency - run the celery daemon with
                        argument 'concurrency'
celeryd view            - view all tasks in the queue
celeryd clean           - delete all tasks in the queue
```

check-po-files: Check po files for common mistakes

Usage:

```
check-po-files [options] [FILE] ...
```

color: Create or remove a color scheme

After running this command, you'll need to regenerate the css files. See *less: Compile all root less documents into their CSS counterparts* for details.

Usage:

```
color                - creates a random color scheme
color clear          - clears any color scheme
color <'HEX'>        - uses as base color eg '#ff00ff' must be quoted.
color <VALUE>        - a float between 0.0 and 1.0 used as base hue
color <COLOR_NAME>   - html color name used for base color eg lightblue
```

create-test-data: Create test data

As the name suggests, this command lets you load test data when first setting up CKAN. See *Creating test data* for details.

dataset: Manage datasets

Usage:

```
dataset DATASET_NAME|ID      - shows dataset properties
dataset show DATASET_NAME|ID - shows dataset properties
dataset list                  - lists datasets
dataset delete [DATASET_NAME|ID] - changes dataset state to 'deleted'
dataset purge [DATASET_NAME|ID] - removes dataset from db entirely
```

datastore: Perform commands to set up the datastore

Make sure that the datastore URLs are set properly before you run these commands.

Usage:

```
datastore set-permissions - shows a SQL script to execute
```

db: Manage databases

Lets you initialise, upgrade, and dump the CKAN database.

Initialization

Before you can run CKAN for the first time, you need to run `db init` to initialize your database:

```
paster db init -c /etc/ckan/default/production.ini
```

If you forget to do this you'll see this error message in your web browser:

503 Service Unavailable: This site is currently off-line. Database is not initialised.

Cleaning

You can delete everything in the CKAN database, including the tables, to start from scratch:

Warning: This will delete all data from your CKAN database!

```
paster db clean -c /etc/ckan/default/production.ini
```

After cleaning the db you must do a `db init` or `db load` before CKAN will work again.

Dumping and Loading databases to/from a file

You can ‘dump’ (save) the exact state of the database to a file on disk and at a later point ‘load’ (restore) it again.

Tip: You can also dump the database from one CKAN instance, and then load it into another CKAN instance on the same or another machine. This will even work if the CKAN instance you dumped the database from is an older version of CKAN than the one you load it into, the database will be automatically upgraded during the load command. (But you cannot load a database from a newer version of CKAN into an older version of CKAN.)

To export a dump of your CKAN database:

```
paster db dump -c /etc/ckan/default/production.ini my_database_dump.sql
```

To load it in again, you first have to clean the database (this will delete all data in the database!) and then load the file:

```
paster db clean -c /etc/ckan/default/production.ini
paster db load -c /etc/ckan/default/production.ini my_database_dump.sql
```

Exporting Datasets to JSON or CSV

You can export all of your CKAN site’s datasets from your database to a JSON file using the `db simple-dump-json` command:

```
paster db simple-dump-json -c /etc/ckan/default/production.ini my_datasets.json
```

To export the datasets in CSV format instead, use `db simple-dump-csv`:

```
paster db simple-dump-csv -c /etc/ckan/default/production.ini my_datasets.csv
```

This is useful to create a simple public listing of the datasets, with no user information. Some simple additions to the Apache config can serve the dump files to users in a directory listing. To do this, add these lines to your virtual Apache config file (e.g. `/etc/apache2/sites-available/ckan_default`):

```
Alias /dump/ /home/okfn/var/srv/ckan.net/dumps/

# Disable the mod_python handler for static files
<Location /dump>
    SetHandler None
    Options +Indexes
</Location>
```

Warning: Don’t serve an SQL dump of your database (created using the `paster db dump` command), as those contain private user information such as email addresses and API keys.

Exporting User Accounts to CSV

You can export all of your CKAN site's user accounts from your database to a CSV file using the `db user-dump-csv` command:

```
paster db user-dump-csv -c /etc/ckan/default/production.ini my_database_users.csv
```

front-end-build: Creates and minifies css and JavaScript files

Usage:

```
front-end-build
```

less: Compile all root less documents into their CSS counterparts

Usage:

```
less
```

minify: Create minified versions of the given Javascript and CSS files

Usage:

```
paster minify [--clean] PATH
```

For example:

```
paster minify ckan/public/base
paster minify ckan/public/base/css/*.css
paster minify ckan/public/base/css/red.css
```

If the `--clean` option is provided any minified files will be removed.

notify: Send out modification notifications

Usage:

```
notify replay    - send out modification signals. In "replay" mode,
                  an update signal is sent for each dataset in the database.
```

plugin-info: Provide info on installed plugins

As the name suggests, this command shows you the installed plugins, their description, and which interfaces they implement

profile: Code speed profiler

Provide a ckan url and it will make the request and record how long each function call took in a file that can be read by `runsnakerun`.

Usage:

profile URL

The result is saved in `profile.data.search`. To view the profile in `runsnakerun`:

```
runsnakerun ckan.data.search.profile
```

You may need to install the `cProfile` python module.

ratings: Manage dataset ratings

Manages the ratings stored in the database, and can be used to count ratings, remove all ratings, or remove only anonymous ratings.

For example, to remove anonymous ratings from the database:

```
paster --plugin=ckan ratings clean-anonymous --config=/etc/ckan/std/std.ini
```

rdf-export: Export datasets as RDF

This command dumps out all currently active datasets as RDF into the specified folder:

```
paster rdf-export /path/to/store/output
```

search-index: Rebuild search index

Rebuilds the search index. This is useful to prevent search indexes from getting out of sync with the main database.

For example:

```
paster --plugin=ckan search-index rebuild --config=/etc/ckan/std/std.ini
```

This default behaviour will clear the index and rebuild it with all datasets. If you want to rebuild it for only one dataset, you can provide a dataset name:

```
paster --plugin=ckan search-index rebuild test-dataset-name --config=/etc/ckan/std/std.ini
```

Alternatively, you can use the `-o` or `--only-missing` option to only reindex datasets which are not already indexed:

```
paster --plugin=ckan search-index rebuild -o --config=/etc/ckan/std/std.ini
```

If you don't want to rebuild the whole index, but just refresh it, use the `-r` or `--refresh` option. This won't clear the index before starting rebuilding it:

```
paster --plugin=ckan search-index rebuild -r --config=/etc/ckan/std/std.ini
```

There is also an option available which works like the refresh option but tries to use all processes on the computer to reindex faster:

```
paster --plugin=ckan search-index rebuild_fast --config=/etc/ckan/std/std.ini
```

There are other search related commands, mostly useful for debugging purposes:

```
search-index check          - checks for datasets not indexed
search-index show DATASET_NAME - shows index of a dataset
search-index clear [DATASET_NAME] - clears the search index for the provided dataset or for the whole index
```

sysadmin: Give sysadmin rights

Gives sysadmin rights to a named user. This means the user can perform any action on any object.

For example, to make a user called ‘admin’ into a sysadmin:

```
paster --plugin=ckan sysadmin add admin --config=/etc/ckan/std/std.ini
```

tracking: Update tracking statistics

Usage:

```
tracking update [start_date]      - update tracking stats
tracking export FILE [start_date] - export tracking stats to a csv file
```

trans: Translation helper functions

Usage:

```
trans js      - generate the javascript translations
trans mangle  - mangle the zh_TW translations for testing
```

user: Create and manage users

Lets you create, remove, list and manage users.

For example, to create a new user called ‘admin’:

```
paster --plugin=ckan user add admin --config=/etc/ckan/std/std.ini
```

To delete the ‘admin’ user:

```
paster --plugin=ckan user remove admin --config=/etc/ckan/std/std.ini
```

Organizations and authorization

Changed in version 2.0: Previous versions of CKAN used a different authorization system.

CKAN’s authorization system controls which users are allowed to carry out which actions on the site. All actions that users can carry out on a CKAN site are controlled by the authorization system. For example, the authorization system controls who can register new user accounts, delete user accounts, or create, edit and delete datasets, groups and organizations.

Authorization in CKAN can be controlled in three ways:

1. Organizations
2. Configuration file options
3. Extensions

The following sections explain each of the three methods in turn.

Note: An **organization admin** in CKAN is an administrator of a particular organization within the site, with control over that organization and its members and datasets. A **sysadmin** is an administrator of the site itself. Sysadmins

can always do everything, including adding, editing and deleting datasets, organizations and groups, regardless of the organization roles and configuration options described below.

Organizations

Organizations are the primary way to control who can see, create and update datasets in CKAN. Each dataset can belong to a single organization, and each organization controls access to its datasets.

Datasets can be marked as public or private. Public datasets are visible to everyone. Private datasets can only be seen by logged-in users who are members of the dataset's organization. Private datasets are not shown in general dataset searches but are shown in dataset searches within the organization.

When a user joins an organization, an organization admin gives them one of three roles: member, editor or admin.

An organization **admin** can:

- View the organization's private datasets
- Add new datasets to the organization
- Edit or delete any of the organization's datasets
- Make datasets public or private.
- Add users to the organization, and choose whether to make the new user a member, editor or admin
- Change the role of any user in the organization, including other admin users
- Remove members, editors or other admins from the organization
- Edit the organization itself (for example: change the organization's title, description or image)
- Delete the organization

An **editor** can:

- View the organization's private datasets
- Add new datasets to the organization
- Edit or delete any of the organization's datasets

A **member** can:

- View the organization's private datasets.

When a user creates a new organization, they automatically become the first admin of that organization.

Configuration File Options

The following configuration file options can be used to customize CKAN's authorization behavior:

ckan.auth.anon_create_dataset

Example:

```
ckan.auth.anon_create_dataset = False
```

Default value: `False`

Allow users to create datasets without registering and logging in.

ckan.auth.create_unowned_dataset

Example:

```
ckan.auth.create_unowned_dataset = False
```

Default value: True

Allow the creation of datasets not owned by any organization.

ckan.auth.create_dataset_if_not_in_organization

Example:

```
ckan.auth.create_dataset_if_not_in_organization = False
```

Default value: True

Allow users who are not members of any organization to create datasets, default: true. create_unowned_dataset must also be True, otherwise setting create_dataset_if_not_in_organization to True is meaningless.

ckan.auth.user_create_groups

Example:

```
ckan.auth.user_create_groups = False
```

Default value: True

Allow users to create groups.

ckan.auth.user_create_organizations

Example:

```
ckan.auth.user_create_organizations = False
```

Default value: True

Allow users to create organizations.

ckan.auth.user_delete_groups

Example:

```
ckan.auth.user_delete_groups = False
```

Default value: True

Allow users to delete groups.

ckan.auth.user_delete_organizations

Example:

```
ckan.auth.user_delete_organizations = False
```

Default value: True

Allow users to delete organizations.

ckan.auth.create_user_via_api

Example:

```
ckan.auth.create_user_via_api = False
```

Default value: False

Allow new user accounts to be created via the API.

ckan.auth.create_user_via_web

Example:

```
ckan.auth.create_user_via_web = True
```

Default value: True

Allow new user accounts to be created via the Web.

ckan.auth.roles_that_cascade_to_sub_groups

Example:

```
ckan.auth.roles_that_cascade_to_sub_groups = admin editor
```

Default value: admin

Makes role permissions apply to all the groups down the hierarchy from the groups that the role is applied to.

e.g. a particular user has the ‘admin’ role for group ‘Department of Health’. If you set the value of this option to ‘admin’ then the user will automatically have the same admin permissions for the child groups of ‘Department of Health’ such as ‘Cancer Research’ (and its children too and so on).

Extensions

CKAN extensions can implement custom authorization rules by overriding the authorization functions that CKAN uses. This is done by implementing the `IAuthFunctions` plugin interface. To get started with writing CKAN extensions, see [Extending guide](#).

Data preview and visualization

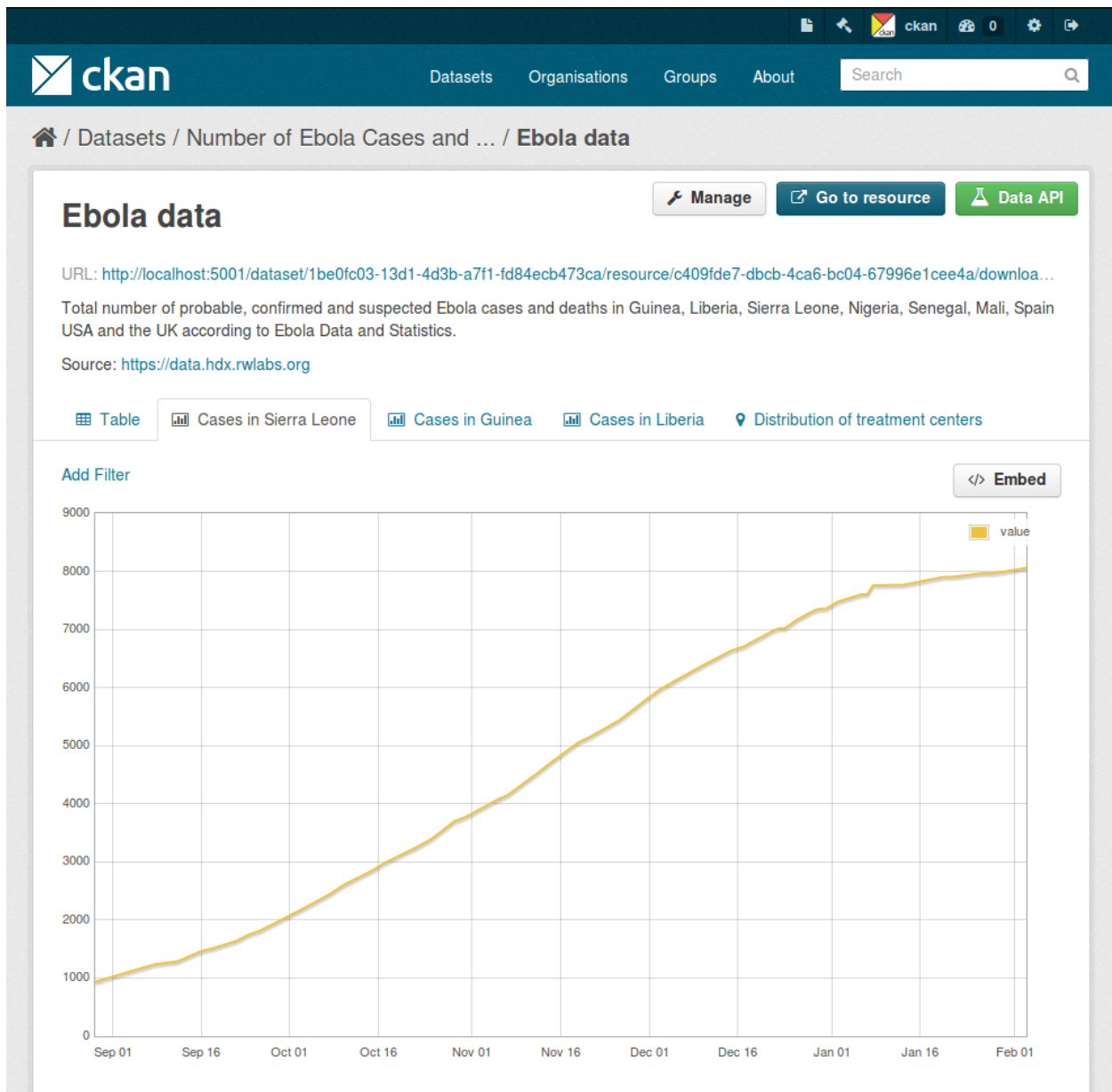
Changed in version 2.3: The whole way resource previews are handled was changed on CKAN 2.3. Please refer to previous versions of the documentation if you are using an older CKAN version.

Contents

- Data preview and visualization
 - Overview
 - Managing resource views
 - Defining views to appear by default
 - Available view plugins
 - * Data Explorer
 - * DataStore Grid
 - * DataStore Graph
 - * DataStore Map
 - * Text view
 - * Image view
 - * Web page view
 - * Other view plugins
 - Resource Proxy
 - Migrating from previous CKAN versions
 - Command line interface

Overview

The CKAN resource page can contain one or more visualizations of the resource data or file contents (a table, a bar chart, a map, etc). These are commonly referred to as *resource views*.



The main features of resource views are:

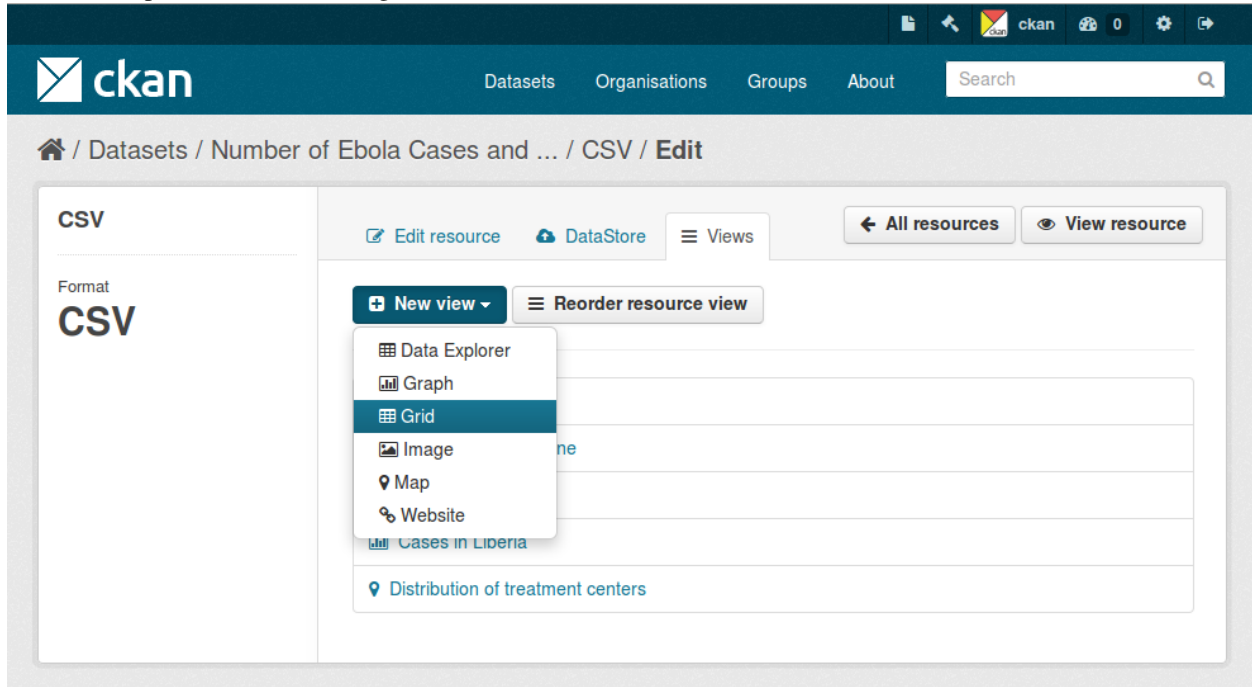
- One resource can have multiple views of the same data (for example a grid and some graphs for tabular data).
- Dataset editors can choose which views to show, reorder them and configure them individually.
- Individual views can be embedded on external sites.

Different view types are implemented via custom plugins, which can be activated on a particular CKAN site. Once these plugins are added, instance administrators can decide which views should be created by default if the resource is suitable (for instance a table on resources uploaded to the DataStore, a map for spatial data, etc.).

Whether a particular resource can be rendered by the different view plugins is decided by the view plugins themselves. This is generally done checking the resource format or whether its data is on the *DataStore extension* or not.

Managing resource views

Users who are allowed to edit a particular dataset can also manage the views for its resources. To access the management interface, click on the *Manage* button on the resource page and then on the *Views* tab. From here you can create new views, update or delete existing ones and reorder them.



The *New view* dropdown will show the available view types for this particular resource. If the list is empty, you may need to add the relevant view plugins to the `ckan.plugins` setting on your configuration file, eg:

```
ckan.plugins = ... image_view recline_view pdf_view
```

Defining views to appear by default

From the management interface you can create and edit views manually, but in most cases you will want views to be created automatically on certain resource types, so data can be visualized straight away after uploading or linking to a file.

To do so, you define a set of view plugins that should be checked whenever a dataset or resource is created or updated. For each of them, if the resource is a suitable one, a view will be created.

This is configured with the `ckan.views.default_views` setting. In it you define the view plugins that you want to be created as default:

```
ckan.views.default_views = recline_view pdf_view geojson_view
```

This configuration does not mean that each new resource will get all of these views by default, but that for instance if the uploaded file is a PDF file, a PDF viewer will be created automatically and so on.

Available view plugins

Some view plugins for common formats are included in the main CKAN repository. These don't require further setup and can be directly added to the `ckan.plugins` setting.

Data Explorer

The screenshot shows the CKAN Data Explorer interface. At the top, there's a 'Data Explorer' tab and an 'Embed' button. Below the tab is an 'Add Filter' link. The main area displays a table with 11 columns: `_id`, `county`, `Murder`, `Rape`, `Robbery`, `Aggrava...`, `Burglary`, `Larceny`, `Motor V...`, `Arson`, and `Populati...`. The table contains 18 rows of data. Above the table, there are controls for view type (Grid, Graph, Map), record count (88 records), pagination (1 - 88), a search bar (Search data ...), and a 'Go »' button. A 'Filters' button is also present.

_id	county	Murder	Rape	Robbery	Aggrava...	Burglary	Larceny	Motor V...	Arson	Populati...
1	Fairfield	14	172	946	1127	2969	10706	1323	52	931568
2	Hartford	33	186	1000	1280	3688	15151	1577	91	904776
3	Litchfield	0	25	24	120	439	2058	144	14	192309
4	Middlesex	2	22	49	144	395	1720	216	35	163682
5	New Hav...	27	125	1352	1604	3576	17163	2506	33	869759
6	New Lon...	6	62	108	446	1159	3375	252	29	273482
7	Tolland	1	20	22	54	315	884	74	7	143932
8	Windham	3	19	27	87	364	1014	123	10	115141
9	Fairfield	34	484	947	1276	3223	11598	1514	63	930450
10	Hartford	35	129	1152	1433	4323	16372	2025	140	903692
11	Litchfield	3	42	24	88	521	2290	133	7	192076
12	Middlesex	29	22	45	136	476	1898	200	42	163485
13	New Hav...	33	140	1355	1850	4295	18372	2127	58	868716
14	New Lon...	7	59	114	490	1056	3520	236	48	273153
15	Tolland	1	25	21	66	370	1028	101	7	143759
16	Windham	1	22	33	71	474	943	105	12	115002
17	Fairfield	32	229	1094	1307	3620	12118	1796	62	927954
18	Hartford	26	168	1098	1522	4581	15805	1821	140	901268

View plugin: `recline_view`

Adds a rich widget, based on the [Recline](#) Javascript library. It allows querying, filtering, graphing and mapping data. The Data Explorer is optimized for displaying structured data hosted on the [DataStore extension](#).

The Data Explorer can also display certain formats of tabular data (CSV and Excel files) without its contents being uploaded to the DataStore. This is done via the [DataProxy](#), an external service that will parse the contents of the file and return a response that the view widget understands. However, as the resource must be downloaded by the DataProxy service and parsed before it is viewed, this option is slower and less reliable than viewing data that is in the DataStore. It also does not properly support different encodings, proper field type detection, etc so users are strongly encouraged to host data on the DataStore instead.

Note: Support for the DataProxy will be dropped on future CKAN releases

The three main panes of the Data Explorer are also available as separate views.

DataStore Grid

View of data within the DataStore </> Embed

Add Filter

« 1 – 100 » 214 records Q Search data ... Go »

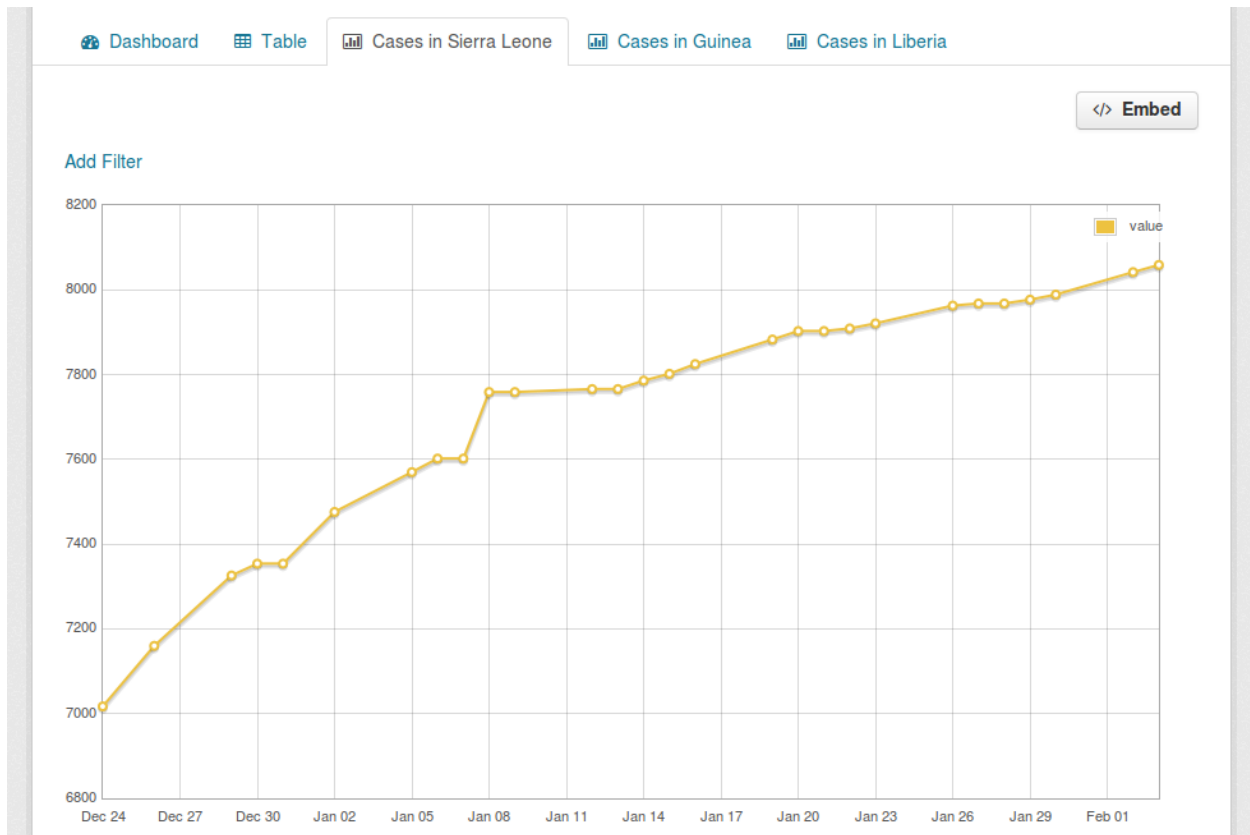
_id	Country...	Country...	Indicato...	Indicato...	2000	2001	2002	2003	2004	2005
1	Afghanis...	AFG	Internet ...	IT.NET....		0.00472...	0.00456...	0.08789...	0.10580...	1.22414...
2	Albania	ALB	Internet ...	IT.NET....	0.11409...	0.32579...	0.39008...	0.97190...	2.42038...	6.04389...
3	Algeria	DZA	Internet ...	IT.NET....	0.49170...	0.64611...	1.59164...	2.19535...	4.63447...	5.84394...
4	America...	ASM	Internet ...	IT.NET....						
5	Andorra	ADO	Internet ...	IT.NET....	10.5388...		11.2604...	13.5464...	26.8379...	37.6057...
6	Angola	AGO	Internet ...	IT.NET....	0.10504...	0.13601...	0.27037...	0.37068...	0.46481...	1.14336...
7	Antigua ...	ATG	Internet ...	IT.NET....	6.48222...	8.89928...	12.5	17.2286...	24.2665...	34.7164...
8	Argentina	ARG	Internet ...	IT.NET....	7.03868...	9.78080...	10.8821...	11.9136...	16.0366...	17.7205...
9	Armenia	ARM	Internet ...	IT.NET....	1.30047...	1.63109...	1.96040...	4.57521...	4.89900...	5.25298...
10	Aruba	ABW	Internet ...	IT.NET....	15.4428...	17.1	18.8	20.8	23	25.4
11	Australia	AUS	Internet ...	IT.NET....	46.7561...	52.6892...				63
12	Austria	AUT	Internet ...	IT.NET....	33.7301...	39.1854...	36.56	42.7	54.28	58
13	Azerbaijan	AZE	Internet ...	IT.NET....	0.14775...	0.30556...	4.99971...			8.03037...
14	Bahama...	BHS	Internet ...	IT.NET....	8	11.8	18	20	22	25
83	Honduras	HND	Internet ...	IT.NET....	1.20385...	1.41528...	2.59740...	4.8	5.6	6.5
15	Bahrain	BHR	Internet ...	IT.NET....	6.15373...	15.0386...	18.0507...	21.5549...	21.4586...	21.3037...
16	Banglad...	BGD	Internet ...	IT.NET....	0.07103...	0.12980...	0.13992...	0.16387...	0.19903...	0.24163...

View plugin: `recline_grid_view`

Displays a filterable, sortable, table view of structured data.

This plugin requires data to be in the DataStore.

DataStore Graph

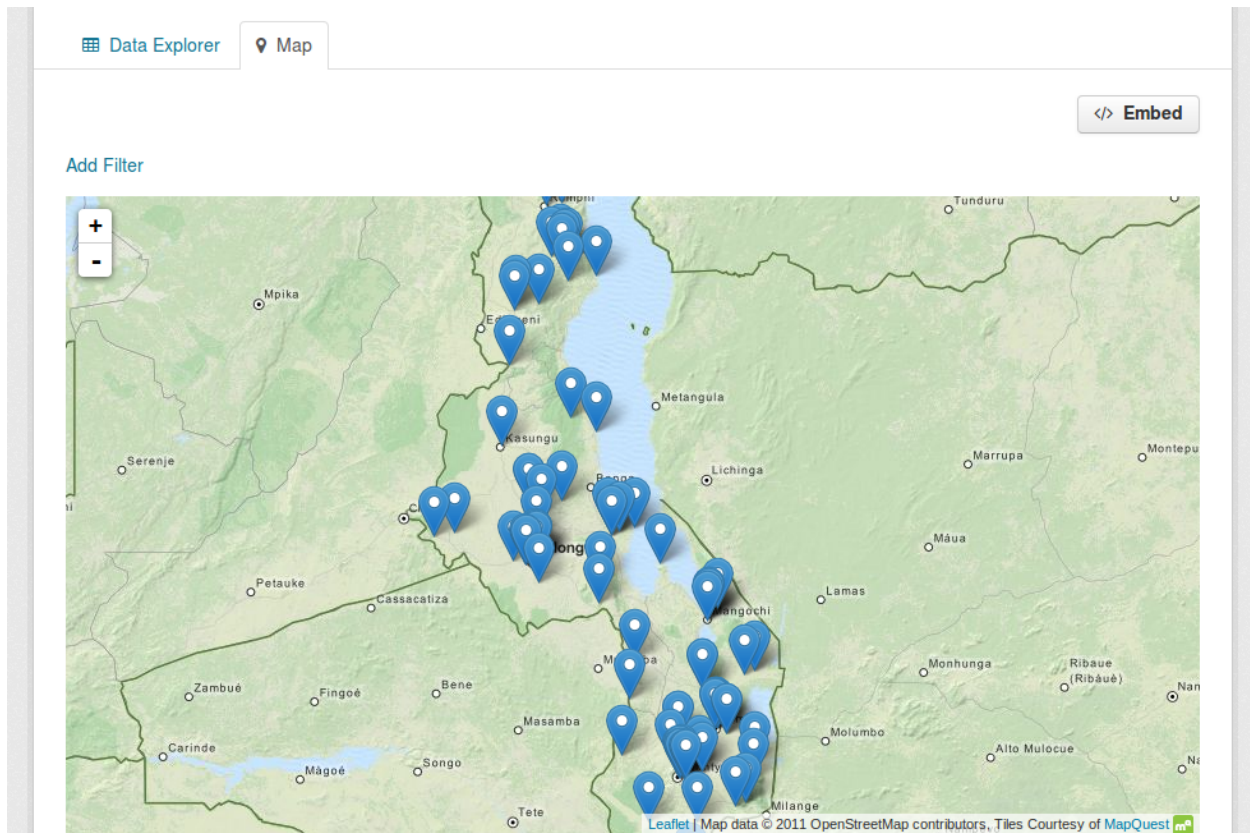


View plugin: `recline_graph_view`

Allows to create graphs from data stored on the DataStore. You can choose the graph type (such as lines, bars, columns, etc) and restrict the displayed data, by filtering by a certain field value or defining an offset and the number of rows.

This plugin requires data to be in the DataStore.

DataStore Map



View plugin: `recline_map_view`

Shows data stored on the DataStore in an interactive map. It supports plotting markers from a pair of latitude / longitude fields or from a field containing a [GeoJSON](#) representation of the geometries. The configuration also allows to cluster markers if there is a high density of them and to zoom automatically to the rendered features.

This plugin requires data to be in the DataStore.

Text view

```
<?xml version="1.0" encoding="UTF-8"?>
<iati-activities generated-datetime="2014-11-24T16:19:39+00:00" version="1.03">
  <!-- Generated By AidStream -->
  <iati-activity xml:lang="en" default-currency="GBP" last-updated-datetime="2014-10-06T16:00:26+00:00">
    <reporting-org ref="GB-CHC-283302" type="21" xml:lang="en">ACORD</reporting-org>
    <iati-identifier>GB-CHC-283302-GDRC08</iati-identifier>
    <title xml:lang="en">Promoting Gender Equality and Food Security in Mushie</title>
    <description xml:lang="en" type="1">This project aims to empower small holder farmers' organisations, esp
    <activity-status code="2"/>
    <activity-date type="start-actual" iso-date="2011-07-01"/>
    <activity-date type="end-planned" iso-date="2014-06-30"/>
    <contact-info>
      <email>info@acordinternational.org</email>
    </contact-info>
    <participating-org xml:lang="en" type="10" role="Funding" ref="GB-1-201242-101">DFID</participating-org>
    <participating-org type="21" role="Accountable" ref="GB-CHC-283302">ACORD</participating-org>
    <participating-org type="21" role="Implementing" ref="GB-CHC-283302">ACORD</participating-org>
    <recipient-country code="CD"/>
    <sector vocabulary="DAC" code="15150"/>
    <default-flow-type code="30"/>
    <default-aid-type code="C01"/>
    <budget type="1">
      <period-start iso-date="2011-07-01"/>
      <period-end iso-date="2014-06-30"/>
      <value currency="GBP" value-date="2011-07-01">476748</value>
    </budget>
    <transaction ref="cl-q1-y2">
      <transaction-type code="IF"/>
      <provider-org ref="GB-1" provider-activity-id="GB-1-201242-101">DFID</provider-org>
      <receiver-org ref="GB-CHC-283302" receiver-activity-id="GB-CHC-283302-GDRC08">ACORD</receiver-org>
      <value currency="GBP" value-date="2012-06-14">31375</value>
      <transaction-date iso-date="2012-06-14"/>
    </transaction>
  </iati-activity>
</iati-activities>
```

View plugin: `text_view`

Displays files in XML, JSON or plain text based formats with the syntax highlighted. The formats detected can be configured using the `ckan.preview.xml_formats`, `ckan.preview.json_formats` and `ckan.preview.text_formats` configuration options respectively.

If you want to display files that are hosted in a different server from your CKAN instance (eg that haven't been uploaded to CKAN) you will need to enable the [Resource Proxy](#) plugin.

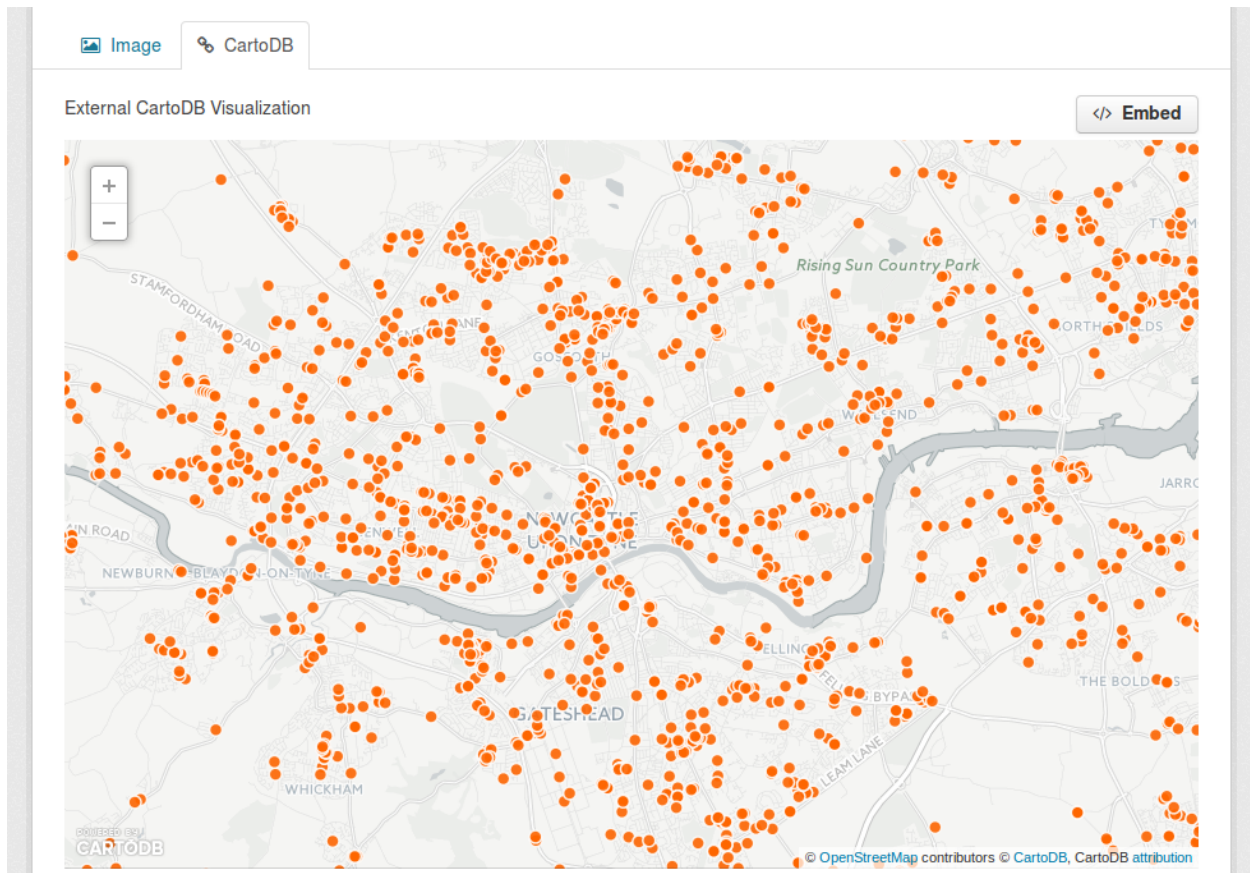
Image view



View plugin: `image_view`

If the resource format is a common image format like PNG, JPEG or GIF, it adds an `` tag pointing to the resource URL. You can provide an alternative URL on the edit view form.

Web page view



View plugin: webpage_view

Adds an `<iframe>` tag to embed the resource URL. You can provide an alternative URL on the edit view form.

Warning: Do not activate this plugin unless you trust the URL sources. It is not recommended to enable this view type on instances where all users can create datasets.

Other view plugins

There are many more view plugins developed by the CKAN team and others which are hosted on separate repositories. Some examples include:

- **Dashboard:** Allows to combine multiple views into a single dashboard.
- **PDF viewer:** Allows to render PDF files on the resource page.
- **GeoJSON map:** Renders GeoJSON files on an interactive map.
- **Choropleth map:** Displays data on the DataStore on a choropleth map.
- **Basic charts:** Provides alternative graph types and renderings.

If you want to add another view type to this list, edit this file by sending a pull request on GitHub.

New plugins to render custom view types can be implemented using the `IResourceView` interface.

Todo

Link to a proper tutorial for writing custom views

Resource Proxy

As resource views are rendered on the browser, if the file they are accessing is located in a different domain than the one CKAN is hosted, the browser will block access to it because of the [same-origin policy](#). For instance, files hosted on *www.example.com* won't be able to be accessed from the browser if CKAN is hosted on *data.catalog.com*.

To allow view plugins access to external files you need to activate the `resource_proxy` plugin on your configuration file:

```
ckan.plugins = resource_proxy ...
```

This will request the file on the server side and serve it from the same domain as CKAN.

You can modify the maximum allowed size for proxied files using the `ckan.resource_proxy.max_file_size` configuration setting.

Migrating from previous CKAN versions

If you are upgrading an existing instance running CKAN version 2.2.x or lower to CKAN 2.3 or higher, you need to perform a migration process in order for the resource views to appear. If the migration does not take place, resource views will only appear when creating or updating datasets or resources, but not on existing ones.

The migration process involves creating the necessary view objects in the database, which can be done using the `paster views create` command.

Note: The `paster views create` command uses the search API to get all necessary datasets and resources, so make sure your search index *is up to date* before starting the migration process.

The way the `paster views create` commands works is getting all or a subset of the instance datasets from the search index, and for each of them checking against a list of view plugins if it is necessary to create a view object. This gets determined by each of the individual view plugins depending on the dataset's resources fields.

Before each run, you will be prompted with the number of datasets affected and asked if you want to continue (unless you pass the `-y` option):

```
You are about to check 3336 datasets for the following view plugins: ['image_view', 'recline_view',  
Do you want to continue? [Y/n]
```

Note: On large CKAN instances the migration process can take a significant time if using the default options. It is worth planning in advance and split the process using the search parameters to only check relevant datasets. The following documentation provides guidance on how to do this.

If no view types are provided, the default ones are used (check [Defining views to appear by default](#) to see how these are defined):

```
paster views create
```

Specific view types can be also provided:

```
paster views create image_view recline_view pdf_view
```


For certain view types (the ones with plugins included in the main CKAN core), default filters are applied to the search to only get relevant resources. For instance if `image_view` is defined, filters are added to the search to only get datasets with resources that have image formats (png, jpg, etc).

You can also provide arbitrary search parameters like the ones supported by `package_search()`. This can be useful for instance to only include datasets with resources of a certain format:

```
paster views create geojson_view -s '{"fq": "res_format:GEOJSON"}'
```

To instead avoid certain formats you can do:

```
paster views create -s '{"fq": "-res_format:HTML"}'
```

Of course this is not limited to resource formats, you can filter out or in using any field, as in a normal dataset search:

```
paster views create -s '{"q": "groups:visualization-examples"}'
```

Tip: If you set the `ckan_logger` level to `DEBUG` on your configuration file you can see the full search parameters being sent to Solr.

For convenience, there is also an option to create views on a particular dataset or datasets:

```
paster views create -d dataset_id
```

```
paster views create -d dataset_name -d dataset_name
```

Command line interface

The `paster views` command allows to create and remove resource views objects from the database in bulk.

Check the command help for the full options:

```
paster views create -h
```

Todo

Tutorial for writing custom view types.

FileStore and file uploads

When enabled, CKAN's FileStore allows users to upload data files to CKAN resources, and to upload logo images for groups and organizations. Users will see an upload button when creating or updating a resource, group or organization.

New in version 2.2: Uploading logo images for groups and organizations was added in CKAN 2.2.

Changed in version 2.2: Previous versions of CKAN used to allow uploads to remote cloud hosting but we have simplified this to only allow local file uploads (see [Migration from 2.1 to 2.2](#) for details on how to migrate). This is to give CKAN more control over the files and make access control possible.

See also:

[DataStore extension](#)

Resource files linked-to from CKAN or uploaded to CKAN's FileStore can also be pushed into CKAN's DataStore, which then enables data previews and a data API for the resources.

Setup file uploads

To setup CKAN's FileStore with local file storage:

1. Create the directory where CKAN will store uploaded files:

```
sudo mkdir -p /var/lib/ckan/default
```

2. Add the following line to your CKAN config file, after the `[app:main]` line:

```
ckan.storage_path = /var/lib/ckan/default
```

3. Set the permissions of your `ckan.storage_path` directory. For example if you're running CKAN with Apache, then Apache's user (`www-data` on Ubuntu) must have read, write and execute permissions for the `ckan.storage_path`:

```
sudo chown www-data /var/lib/ckan/default
sudo chmod u+rwX /var/lib/ckan/default
```

4. Restart your web server, for example to restart Apache:

```
sudo service apache2 reload
```

FileStore API

Changed in version 2.2: The FileStore API was redesigned for CKAN 2.2. The previous API has been deprecated.

Files can be uploaded to the FileStore using the `resource_create()` and `resource_update()` action API functions. You can post multipart/form-data to the API and the key, value pairs will be treated as if they are a JSON object. The extra key `upload` is used to actually post the binary data.

For example, to create a new CKAN resource and upload a file to it using `curl`:

```
curl -H'Authorization: your-api-key' 'http://yourhost/api/action/resource_create' --form upload=@file
```

(Curl automatically sends a `multipart-form-data` heading with you use the `--form` option.)

To create a new resource and upload a file to it using the Python library `requests`:

```
import requests
requests.post('http://0.0.0.0:5000/api/action/resource_create',
              data={"package_id": "my_dataset"},
              headers={"X-CKAN-API-Key": "21a47217-6d7b-49c5-88f9-72ebd5a4d4bb"},
              files=[('upload', file('/path/to/file/to/upload.csv'))])
```

(Requests automatically sends a `multipart-form-data` heading when you use the `files=` parameter.)

To overwrite an uploaded file with a new version of the file, post to the `resource_update()` action and use the `upload` field:

```
curl -H'Authorization: your-api-key' 'http://yourhost/api/action/resource_update' --form upload=@newfile
```

To replace an uploaded file with a link to a file at a remote URL, use the `clear_upload` field:

```
curl -H'Authorization: your-api-key' 'http://yourhost/api/action/resource_update' --form url=http://remoteurl
```

Migration from 2.1 to 2.2

If you are using pairtree local file storage then you can keep your current settings without issue. The pairtree and new storage can live side by side but you are still encouraged to migrate. If you change your config options to the ones specified in this docs you will need to run the migration below.

If you are running remote storage then all previous links will still be accessible but if you want to move the remote storage documents to the local storage you will run the migration also.

In order to migrate make sure your CKAN instance is running as the script will request the data from the instance using APIs. You need to run the following on the command line to do the migration:

```
paster db migrate-filestore
```

This may take a long time especially if you have a lot of files remotely. If the remote hosting goes down or the job is interrupted it is save to run it again and it will try all the unsuccessful ones again.

Custom Internet media types (MIME types)

New in version 2.2.

CKAN uses the default Python library [mimetypes](#) to detect the media type of an uploaded file. If some particular format is not included in the ones guessed by the [mimetypes](#) library, a default `application/octet-stream` value will be returned.

Users can still register a more appropriate media type by using the [mimetypes](#) library. A good way to do so is to use the `IConfigurer` interface so the custom types get registered on startup:

```
import mimetypes
import ckan.plugins as p

class MyPlugin(p.SingletonPlugin):

    p.implements(p.IConfigurer)

    def update_config(self, config):

        mimetypes.add_type('application/json', '.geojson')

        # ...
```

DataStore extension

The CKAN DataStore extension provides an *ad hoc* database for storage of structured data from CKAN resources. Data can be pulled out of resource files and stored in the DataStore.

When a resource is added to the DataStore, you get:

- Automatic data previews on the resource's page, using the [Data Explorer extension](#)
- The [DataStore API](#): search, filter and update the data, without having to download and upload the entire data file

The DataStore is integrated into the [CKAN API](#) and authorization system.

The DataStore is generally used alongside the [DataPusher](#), which will automatically upload data to the DataStore from suitable files, whether uploaded to CKAN's FileStore or externally linked.

- [Relationship to FileStore](#)
- [Setting up the DataStore](#)
- [DataPusher: Automatically Add Data to the DataStore](#)
- [The DataStore API](#)

Relationship to FileStore

The DataStore is distinct but complementary to the FileStore (see *FileStore and file uploads*). In contrast to the FileStore which provides ‘blob’ storage of whole files with no way to access or query parts of that file, the DataStore is like a database in which individual data elements are accessible and queryable. To illustrate this distinction, consider storing a spreadsheet file like a CSV or Excel document. In the FileStore this file would be stored directly. To access it you would download the file as a whole. By contrast, if the spreadsheet data is stored in the DataStore, one would be able to access individual spreadsheet rows via a simple web API, as well as being able to make queries over the spreadsheet contents.

Setting up the DataStore

Note: The DataStore requires PostgreSQL 9.0 or later. It is possible to use the DataStore on versions prior to 9.0 (for example 8.4). However, the `datastore_search_sql()` will not be available and the set-up is slightly different. Make sure, you read *Legacy mode: use the DataStore with old PostgreSQL versions* for more details.

1. Enable the plugin

Add the datastore plugin to your CKAN config file:

```
ckan.plugins = datastore
```

2. Set-up the database

Warning: Make sure that you follow the steps in [Set Permissions](#) below correctly. Wrong settings could lead to serious security issues.

The DataStore requires a separate PostgreSQL database to save the DataStore resources to.

List existing databases:

```
sudo -u postgres psql -l
```

Check that the encoding of databases is UTF8, if not internationalisation may be a problem. Since changing the encoding of PostgreSQL may mean deleting existing databases, it is suggested that this is fixed before continuing with the datastore setup.

Create users and databases

Tip: If your CKAN database and DataStore databases are on different servers, then you need to create a new database user on the server where the DataStore database will be created. As in *Installing CKAN from source* we’ll name the database user `ckan_default`:

```
sudo -u postgres createuser -S -D -R -P -l ckan_default
```

Create a database_user called `datastore_default`. This user will be given read-only access to your DataStore database in the [Set Permissions](#) step below:

```
sudo -u postgres createuser -S -D -R -P -l datastore_default
```

Create the database (owned by `ckan_default`), which we’ll call `datastore_default`:

```
sudo -u postgres createdb -O ckan_default datastore_default -E utf-8
```

Set URLs

Now, uncomment the `ckan.datastore.write_url` and `ckan.datastore.read_url` lines in your CKAN config file and edit them if necessary, for example:

```
ckan.datastore.write_url = postgresql://ckan_default:pass@localhost/datastore_default
ckan.datastore.read_url = postgresql://datastore_default:pass@localhost/datastore_default
```

Replace `pass` with the passwords you created for your `ckan_default` and `datastore_default` database users.

Set permissions

Tip: See *Legacy mode: use the DataStore with old PostgreSQL versions* if these steps continue to fail or seem too complicated for your set-up. However, keep in mind that the legacy mode is limited in its capabilities.

Once the DataStore database and the users are created, the permissions on the DataStore and CKAN database have to be set. CKAN provides a paster command to help you correctly set these permissions.

If you are able to use the `psql` command to connect to your database as a superuser, you can use the `datastore set-permissions` command to emit the appropriate SQL to set the permissions.

For example, if you can connect to your database server as the `postgres` superuser using:

```
sudo -u postgres psql
```

Then you can use this connection to set the permissions:

```
sudo ckan datastore set-permissions |
sudo -u postgres psql --set ON_ERROR_STOP=1
```

Note: If you performed a source install, you will need to replace all references to `sudo ckan ...` with `paster --plugin=ckan ...` and provide the path to the config file, e.g. `paster --plugin=ckan datastore set-permissions -c /etc/ckan/default/development.ini`

If your database server is not local, but you can access it over SSH, you can pipe the permissions script over SSH:

```
sudo ckan datastore set-permissions |
ssh dbserver sudo -u postgres psql --set ON_ERROR_STOP=1
```

If you can't use the `psql` command in this way, you can simply copy and paste the output of:

```
sudo ckan datastore set-permissions
```

into a PostgreSQL superuser console.

3. Test the set-up

The DataStore is now set-up. To test the set-up, (re)start CKAN and run the following command to list all DataStore resources:

```
curl -X GET "http://127.0.0.1:5000/api/3/action/datastore_search?resource_id=table_metadata"
```

This should return a JSON page without errors.

To test whether the set-up allows writing, you can create a new DataStore resource. To do so, run the following command:

```
curl -X POST http://127.0.0.1:5000/api/3/action/datastore_create -H "Authorization: {YOUR-API-KEY}"
```

Replace {YOUR-API-KEY} with a valid API key and {PACKAGE-ID} with the id of an existing CKAN dataset.

A table named after the resource id should have been created on your DataStore database. Visiting this URL should return a response from the DataStore with the records inserted above:

```
http://127.0.0.1:5000/api/3/action/datastore_search?resource_id={RESOURCE_ID}
```

Replace {RESOURCE-ID} with the resource id that was returned as part of the response of the previous API call.

You can now delete the DataStore table with:

```
curl -X POST http://127.0.0.1:5000/api/3/action/datastore_delete -H "Authorization: {YOUR-API-KEY}"
```

To find out more about the DataStore API, see [The DataStore API](#).

Legacy mode: use the DataStore with old PostgreSQL versions

Tip: The legacy mode can also be used to simplify the set-up since it does not require you to set the permissions or create a separate user.

The DataStore can be used with a PostgreSQL version prior to 9.0 in *legacy mode*. Due to the lack of some functionality, the `datastore_search_sql()` and consequently the *HTSQL support* cannot be used. To enable the legacy mode, remove the declaration of the `ckan.datastore.read_url`.

The set-up for legacy mode is analogous to the normal set-up as described above with a few changes and consists of the following steps:

1. Enable the plugin
2. The legacy mode is enabled by **not** setting the `ckan.datastore.read_url`
3. Set-Up the database
 - (a) Create a separate database
 - (b) Create a write user on the DataStore database (optional since the CKAN user can be used)
4. Test the set-up

There is no need for a read-only user or special permissions. Therefore the legacy mode can be used for simple set-ups as well.

DataPusher: Automatically Add Data to the DataStore

Often, one wants data that is added to CKAN (whether it is linked to or uploaded to the *FileStore*) to be automatically added to the DataStore. This requires some processing, to extract the data from your files and to add it to the DataStore in the format the DataStore can handle.

This task of automatically parsing and then adding data to the DataStore is performed by the *DataPusher*, a service that runs asynchronously and can be installed alongside CKAN.

To install this please look at the docs here: <http://docs.ckan.org/projects/datapusher>

The DataStore API

The CKAN DataStore offers an API for reading, searching and filtering data without the need to download the entire file first. The DataStore is an ad hoc database which means that it is a collection of tables with unknown relationships. This allows you to search in one DataStore resource (a *table* in the database) as well as queries across DataStore resources.

Data can be written incrementally to the DataStore through the API. New data can be inserted, existing data can be updated or deleted. You can also add a new column to an existing table even if the DataStore resource already contains some data.

You will notice that we tried to keep the layer between the underlying PostgreSQL database and the API as thin as possible to allow you to use the features you would expect from a powerful database management system.

A DataStore resource can not be created on its own. It is always required to have an associated CKAN resource. If data is stored in the DataStore, it will automatically be previewed by the [recline preview extension](#).

Making a DataStore API request

Making a DataStore API request is the same as making an Action API request: you post a JSON dictionary in an HTTP POST request to an API URL, and the API also returns its response in a JSON dictionary. See the [API guide](#) for details.

API reference

Note: Lists can always be expressed in different ways. It is possible to use lists, comma separated strings or single items. These are valid lists: `['foo', 'bar']`, `'foo, bar'`, `"foo"`, `"bar"` and `'foo'`. Additionally, there are several ways to define a boolean value. `True`, `on` and `1` are all valid boolean values.

Note: The table structure of the DataStore is explained in [Internal structure of the database](#).

`ckanext.datastore.logic.action.datastore_create(context, data_dict)`

Adds a new table to the DataStore.

The `datastore_create` action allows you to post JSON data to be stored against a resource. This endpoint also supports altering tables, aliases and indexes and bulk insertion. This endpoint can be called multiple times to initially insert more data, add fields, change the aliases or indexes as well as the primary keys.

To create an empty datastore resource and a CKAN resource at the same time, provide `resource` with a valid `package_id` and omit the `resource_id`.

If you want to create a datastore resource from the content of a file, provide `resource` with a valid `url`.

See [Fields](#) and [Records](#) for details on how to lay out records.

Parameters

- **resource_id** (*string*) – resource id that the data is going to be stored against.
- **force** (*bool (optional, default: False)*) – set to `True` to edit a read-only resource
- **resource** (*dictionary*) – resource dictionary that is passed to `resource_create()`. Use instead of `resource_id` (optional)
- **aliases** (*list or comma separated string*) – names for read only aliases of the resource. (optional)
- **fields** (*list of dictionaries*) – fields/columns and their extra metadata. (optional)

- **records** (*list of dictionaries*) – the data, eg: [{"dob": "2005", "some_stuff": ["a", "b"]}](optional)
- **primary_key** (*list or comma separated string*) – fields that represent a unique key (optional)
- **indexes** (*list or comma separated string*) – indexes on table (optional)

Please note that setting the `aliases`, `indexes` or `primary_key` replaces the existing aliases or constraints. Setting `records` appends the provided records to the resource.

Results:

Returns The newly created data object.

Return type dictionary

See [Fields](#) and [Records](#) for details on how to lay out records.

`ckanext.datastore.logic.action.datastore_upsert` (*context, data_dict*)

Updates or inserts into a table in the DataStore

The `datastore_upsert` API action allows you to add or edit records to an existing DataStore resource. In order for the *upsert* and *update* methods to work, a unique key has to be defined via the `datastore_create` action. The available methods are:

upsert Update if record with same key already exists, otherwise insert. Requires unique key.

insert Insert only. This method is faster than upsert, but will fail if any inserted record matches an existing one. Does *not* require a unique key.

update Update only. An exception will occur if the key that should be updated does not exist. Requires unique key.

Parameters

- **resource_id** (*string*) – resource id that the data is going to be stored under.
- **force** (*bool (optional, default: False)*) – set to True to edit a read-only resource
- **records** (*list of dictionaries*) – the data, eg: [{"dob": "2005", "some_stuff": ["a","b"]}](optional)
- **method** (*string*) – the method to use to put the data into the datastore. Possible options are: `upsert`, `insert`, `update` (optional, default: `upsert`)

Results:

Returns The modified data object.

Return type dictionary

`ckanext.datastore.logic.action.datastore_info` (*context, data_dict*)

Returns information about the data imported, such as column names and types.

Return type A dictionary describing the columns and their types.

Parameters **id** (*A UUID*) – Id of the resource we want info about

`ckanext.datastore.logic.action.datastore_delete` (*context, data_dict*)

Deletes a table or a set of records from the DataStore.

Parameters

- **resource_id** (*string*) – resource id that the data will be deleted from. (optional)
- **force** (*bool (optional, default: False)*) – set to True to edit a read-only resource

- **filters** (*dictionary*) – filters to apply before deleting (eg {"name": "fred"}). If missing delete whole table and all dependent views. (optional)

Results:

Returns Original filters sent.

Return type dictionary

`ckanext.datastore.logic.action.datastore_search(context, data_dict)`

Search a DataStore resource.

The `datastore_search` action allows you to search data in a resource. DataStore resources that belong to private CKAN resource can only be read by you if you have access to the CKAN resource and send the appropriate authorization.

Parameters

- **resource_id** (*string*) – id or alias of the resource to be searched against
- **filters** (*dictionary*) – matching conditions to select, e.g {"key1": "a", "key2": "b"} (optional)
- **q** (*string or dictionary*) – full text query. If it's a string, it'll search on all fields on each row. If it's a dictionary as {"key1": "a", "key2": "b"}, it'll search on each specific field (optional)
- **distinct** (*bool*) – return only distinct rows (optional, default: false)
- **plain** (*bool*) – treat as plain text query (optional, default: true)
- **language** (*string*) – language of the full text query (optional, default: english)
- **limit** (*int*) – maximum number of rows to return (optional, default: 100)
- **offset** (*int*) – offset this number of rows (optional)
- **fields** (*list or comma separated string*) – fields to return (optional, default: all fields in original order)
- **sort** (*string*) – comma separated field names with ordering e.g.: "fieldname1, fieldname2 desc"

Setting the `plain` flag to false enables the entire PostgreSQL [full text search query language](#).

A listing of all available resources can be found at the alias `_table_metadata`.

If you need to download the full resource, read [Download resource as CSV](#).

Results:

The result of this action is a dictionary with the following keys:

Return type A dictionary with the following keys

Parameters

- **fields** (*list of dictionaries*) – fields/columns and their extra metadata
- **offset** (*int*) – query offset value
- **limit** (*int*) – query limit value
- **filters** (*list of dictionaries*) – query filters
- **total** (*int*) – number of total matching records
- **records** (*list of dictionaries*) – list of matching results

`ckanext.datastore.logic.action.datastore_search_sql(context, data_dict)`

Execute SQL queries on the DataStore.

The `datastore_search_sql` action allows a user to search data in a resource or connect multiple resources with join expressions. The underlying SQL engine is the [PostgreSQL engine](#). There is an enforced timeout on SQL queries to avoid an unintended DOS. DataStore resource that belong to a private CKAN resource cannot be searched with this action. Use `datastore_search()` instead.

Note: This action is only available when using PostgreSQL 9.X and using a read-only user on the database. It is not available in *legacy mode*.

Parameters `sql` (*string*) – a single SQL select statement

Results:

The result of this action is a dictionary with the following keys:

Return type A dictionary with the following keys

Parameters

- **fields** (*list of dictionaries*) – fields/columns and their extra metadata
- **records** (*list of dictionaries*) – list of matching results

`ckanext.datastore.logic.action.datastore_make_private(context, data_dict)`

Deny access to the DataStore table through `datastore_search_sql()`.

This action is called automatically when a CKAN dataset becomes private or a new DataStore table is created for a CKAN resource that belongs to a private dataset.

Parameters `resource_id` (*string*) – id of resource that should become private

`ckanext.datastore.logic.action.datastore_make_public(context, data_dict)`

Allow access to the DataStore table through `datastore_search_sql()`.

This action is called automatically when a CKAN dataset becomes public.

Parameters `resource_id` (*string*) – id of resource that should become public

Download resource as CSV

A DataStore resource can be downloaded in the CSV file format from `{CKAN-URL}/datastore/dump/{RESOURCE-ID}`.

Fields

Fields define the column names and the type of the data in a column. A field is defined as follows:

```
{
  "id":    # a string which defines the column name
  "type":  # the data type for the column
}
```

Field **types are optional** and will be guessed by the DataStore from the provided data. However, setting the types ensures that future inserts will not fail because of wrong types. See [Field types](#) for details on which types are valid.

Example:


```
[
  {
    "id": "foo",
    "type": "int4"
  },
  {
    "id": "bar"
    # type is optional
  }
]
```

Records

A record is the data to be inserted in a DataStore resource and is defined as follows:

```
{
  "<id>": # data to be set
  # .. more data
}
```

Example:

```
[
  {
    "foo": 100,
    "bar": "Here's some text"
  },
  {
    "foo": 42
  }
]
```

Field types

The DataStore supports all types supported by PostgreSQL as well as a few additions. A list of the PostgreSQL types can be found in the [type section of the documentation](#). Below you can find a list of the most common data types. The `json` type has been added as a storage for nested data.

In addition to the listed types below, you can also use array types. They are defined by prepending a `_` or appending `[]` or `[n]` where `n` denotes the length of the array. An arbitrarily long array of integers would be defined as `int[]`.

text Arbitrary text data, e.g. `Here's some text`.

json Arbitrary nested json data, e.g. `{"foo": 42, "bar": [1, 2, 3]}`. Please note that this type is a custom type that is wrapped by the DataStore.

date Date without time, e.g. `2012-5-25`.

time Time without date, e.g. `12:42`.

timestamp Date and time, e.g. `2012-10-01T02:43Z`.

int Integer numbers, e.g. `42`, `7`.

float Floats, e.g. `1.61803`.

bool Boolean values, e.g. `true`, `0`

You can find more information about the formatting of dates in the [date/time types](#) section of the PostgreSQL documentation.

Resource aliases

A resource in the DataStore can have multiple aliases that are easier to remember than the resource id. Aliases can be created and edited with the `datastore_create()` API endpoint. All aliases can be found in a special view called `_table_metadata`. See [Internal structure of the database](#) for full reference.

HTSQL support

The `ckanext-htsql` extension adds an API action that allows a user to search data in a resource using the HTSQL query expression language. Please refer to the extension documentation to know more.

Comparison of different querying methods

The DataStore supports querying with multiple API endpoints. They are similar but support different features. The following list gives an overview of the different methods.

	<code>datastore_search()</code>	<code>datastore_search_sql()</code>	<i>HTSQL</i>
Ease of use	Easy	Complex	Medium
Flexibility	Low	High	Medium
Query language	Custom (JSON)	SQL	HTSQL
Join resources	No	Yes	No

Internal structure of the database

The DataStore is a thin layer on top of a PostgreSQL database. Each DataStore resource belongs to a CKAN resource. The name of a table in the DataStore is always the resource id of the CKAN resource for the data.

As explained in [Resource aliases](#), a resource can have mnemonic aliases which are stored as views in the database.

All aliases (views) and resources (tables respectively relations) of the DataStore can be found in a special view called `_table_metadata`. To access the list, open `http://{YOUR-CKAN-INSTALLATION}/api/3/action/datastore_search?resource_id=_table_metadata`.

`_table_metadata` has the following fields:

_id Unique key of the relation in `_table_metadata`.

alias_of Name of a relation that this alias point to. This field is `null` iff the name is not an alias.

name Contains the name of the alias if `alias_of` is not null. Otherwise, this is the resource id of the CKAN resource for the DataStore resource.

oid The PostgreSQL object ID of the table that belongs to name.

Apps & Ideas

Since 1.7 CKAN has a feature called Apps & Ideas which allows users to provide information on apps, ideas, visualizations, articles etc that are related to a specific dataset. Once created these items will be shown against the dataset but also shown on the apps dashboard which will allow users to filter the results based on popularity, or type, or the data when the items were created.

This feature is enabled by default but can be disabled using the `ckan.dataset.show_apps_ideas` setting to hide the tab on the dataset pages.

Tag Vocabularies

New in version 1.7.

CKAN sites can have *tag vocabularies*, which are a way of grouping related tags together into custom fields.

For example, if you were making a site for music datasets, you might use a tag vocabulary to add two fields *Genre* and *Composer* to your site's datasets, where each dataset can have one of the values *Avant-Garde*, *Country* or *Jazz* in its genre field, and one of the values *Beethoven*, *Wagner*, or *Tchaikovsky* in its composer field. In this example, genre and composer would be vocabularies and the values would be tags:

- Vocabulary: Genre
 - Tag: Avant-Garde
 - Tag: Country
 - Tag: Jazz
- Vocabulary: Composer
 - Tag: Beethoven
 - Tag: Wagner
 - Tag: Tchaikovsky

Ofcourse, you could just add Avant-Garde, Beethoven, etc. to datasets as normal CKAN tags, but using tag vocabularies lets you define Avant-Garde, Country and Jazz as genres and Beethoven, Wagner and Tchaikovsky as composers, and lets you enforce restrictions such as that each dataset must have a genre and a composer, and that no dataset can have two genres or two composers, etc.

Another example use-case for tag vocabularies would be to add a *Country Code* field to datasets defining the geographical coverage of the dataset, where each dataset is assigned a country code such as *en*, *fr*, *de*, etc. See `ckanext/example_idatasetform` for a working example implementation of country codes as a tag vocabulary.

Properties of Tag Vocabularies

- A CKAN website can have any number of vocabularies.
- Each vocabulary has an ID and name.
- Each tag either belongs to a vocabulary, or can be a *free tag* that doesn't belong to any vocabulary (i.e. a normal CKAN tag).
- A dataset can have more than one tag from the same vocabulary, and can have tags from more than one vocabulary.

Using Vocabularies

To add a tag vocabulary to a site, a CKAN sysadmin must:

1. Call the `vocabulary_create()` action of the CKAN API to create the vocabulary and tags. See [API guide](#).

2. Implement an `IDatasetForm` plugin to add a new field for the tag vocabulary to the dataset schema. See *Extending guide*.
3. Provide custom dataset templates to display the new field to users when adding, updating or viewing datasets in the CKAN web interface. See *Theming guide*.

See `ckanext/example_idatasetform` for a working example of these steps.

Form Integration

CKAN allows you to integrate its Edit Dataset and New Dataset forms into an external front-end. To that end, CKAN also provides a simple way to redirect these forms back to the external front-end upon submission.

Redirecting CKAN Forms

It is obviously simple enough for an external front-end to link to CKAN's Edit Dataset and New Dataset forms, but once the forms are submitted, it would be desirable to redirect the user back to the external front-end, rather than CKAN's dataset read page.

This is achieved with a parameter to the CKAN URL. The 'return URL' can be specified in two places:

1. Passed as a URL-encoded value with the parameter `return_to` in the link to CKAN's form page.
2. Specified in the CKAN config keys `package_new_return_url` and `package_edit_return_url`.

(If the 'return URL' is supplied in both places, then the first takes precedence.)

Since the 'return URL' may need to include the dataset name, which could be changed by the user, CKAN replaces a known placeholder `<NAME>` with this value on redirect.

Note: Note that the downside of specifying the 'return URL' in the CKAN config is that the CKAN web interface becomes less usable on its own, since the user is hampered by the redirects to the external interface.

Example

An external front-end displays a dataset 'ontariolandcoverv100' here:

`http://datadotgc.ca/dataset/ontariolandcoverv100`

It displays a link to edit this dataset using CKAN's form, which without the redirect would be:

`http://ca.ckan.net/dataset/edit/ontariolandcoverv100`

At first, it may seem that the return link should be `http://datadotgc.ca/dataset/ontariolandcoverv100`. But when the user edits this dataset, the name may change. So the return link needs to be:

`http://datadotgc.ca/dataset/<NAME>`

And this is URL-encoded to become:

`http%3A%2F%2Fdatadotgc.ca%2Fdataset%2F%3CNAME%3E`

So, in summary, the edit link becomes:

`http://ca.ckan.net/dataset/edit/ontariolandcoverv100?return_to=http%3A%2F%2Fdatadotgc.ca%2Fdataset%2F%3CNAME%3E`

During editing the dataset, the user changes the dataset name to *canadalandcover*, presses ‘preview’ and finally ‘commit’. The user is now redirected back to the external front-end at:

```
http://datadotgc.ca/dataset/canadalandcover
```

The same functionality could be achieved by this line in the config file (`ca.ckan.net.ini`):

```
...

[app:main]
package_edit_return_url = http://datadotgc.ca/dataset/<NAME>

...
```

Linked Data and RDF

CKAN has extensive support for linked data and RDF. In particular, there is complete and functional mapping of the CKAN dataset schema to linked data formats.

Enabling and Configuring Linked Data Support

In CKAN \leq 1.6 please install the RDF extension: <https://github.com/okfn/ckanext-rdf>

In CKAN \geq 1.7, basic RDF support will be available directly in core.

Configuration

When using the built-in RDF support (CKAN \geq 1.7) there is no configuration required. By default requests for RDF data will return the RDF generated from the built-in ‘packages/read.rdf’ template, which can be overridden using the extra-templates directive.

Accessing Linked Data

To access linked data versions just access the [API guide](#) in the usual way but set the Accept header to the format you would like to be returned. For example:

```
curl -L -H "Accept: application/rdf+xml" http://thedatahub.org/dataset/gold-prices
curl -L -H "Accept: text/n3" http://thedatahub.org/dataset/gold-prices
```

An alternative method of retrieving the data is to add `.rdf` to the name of the dataset to download:

```
curl -L http://thedatahub.org/dataset/gold-prices.rdf
curl -L http://thedatahub.org/dataset/gold-prices.n3
```

Schema Mapping

There are various vocabularies that can be used for describing datasets:

- Dublin core: these are the most well-known and basic. Dublin core terms includes the class *dct:Dataset*.
- **DCAT** - vocabulary for catalogues of datasets
- **VoID** - vocabulary of interlinked datasets. Specifically designed for describing *rdf* datasets. Perfect except for the fact that it is focused on RDF

- **SCOVO**: this is more oriented to statistical datasets but has a *scovo:Dataset* class.

At the present CKAN uses mostly DCAT and Dublin Core.

An example schema might look like:

```
<rdf:RDF xmlns:foaf="http://xmlns.com/foaf/0.1/" xmlns:owl="http://www.w3.org/2002/07/owl#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:dcats="http://www.w3.org/ns/dcat#"
  xmlns:dct="http://purl.org/dc/terms/">
<dcat:Dataset rdf:about="http://127.0.0.1:5000/dataset/worldwide-shark-attacks">
  <owl:sameAs rdf:resource="urn:uuid:424bdc8c-038d-4b44-8f1d-01227e920b69"></owl:sameAs>
  <dct:description>Shark attacks worldwide</dct:description>
  <dcat:keyword>sharks</dcat:keyword>
  <dcat:keyword>worldwide</dcat:keyword>
  <foaf:homepage rdf:resource="http://127.0.0.1:5000/dataset/worldwide-shark-attacks"></foaf:homepage>
  <rdfs:label>worldwide-shark-attacks</rdfs:label>
  <dct:identifier>worldwide-shark-attacks</dct:identifier>
  <dct:title>Worldwide Shark Attacks</dct:title>
  <dcat:distribution>
    <dcat:Distribution>
      <dcat:accessURL rdf:resource="https://api.scrapewiki.com/api/1.0/datastore/sqlite?format=">
    </dcat:Distribution>
  </dcat:distribution>
  <dcat:distribution>
    <dcat:Distribution>
      <dcat:accessURL rdf:resource="https://api.scrapewiki.com/api/1.0/datastore/sqlite?format=">
    </dcat:Distribution>
  </dcat:distribution>
  <dct:creator>
    <rdf:Description>
      <foaf:name>Ross</foaf:name>
      <foaf:mbox rdf:resource="mailto:ross.jones@okfn.org"></foaf:mbox>
    </rdf:Description>
  </dct:creator>
  <dct:contributor>
    <rdf:Description>
      <foaf:name>Ross</foaf:name>
      <foaf:mbox rdf:resource="mailto:ross.jones@okfn.org"></foaf:mbox>
    </rdf:Description>
  </dct:contributor>
  <dct:rights rdf:resource="http://www.opendefinition.org/licenses/odc-pddl"></dct:rights>
</dcat:Dataset>
</rdf:RDF>
```

Background tasks

CKAN allows you to create tasks that run in the ‘background’, that is asynchronously and without blocking the main application (these tasks can also be automatically retried in the case of transient failures). Such tasks can be created in [Extensions](#) or in core CKAN.

Background tasks can be essential to providing certain kinds of functionality, for example:

- Creating webhooks that notify other services when certain changes occur (for example a dataset is updated)
- Performing processing or validation on data (as done by the Archiver and DataStorer Extensions)

Enabling background tasks

To manage and run background tasks requires a job queue and CKAN uses `celery` (plus the CKAN database) for this purpose. Thus, to use background tasks you need to install and run `celery`.

Installation of `celery` will normally be taken care of by whichever component or extension utilizes it so we skip that here.

To run the celery daemon you have two options:

1. In development setup you can just use paster. This can be done as simply as:

```
paster celeryd
```

This only works if you have a `development.ini` file in ckan root.

2. In production, the daemon should be run with a different ini file and be run as an init script. The simplest way to do this is to install supervisor:

```
apt-get install supervisor
```

Using this file as a template and copy to `/etc/supervisor/conf.d:`

```
https://github.com/ckan/ckan/blob/master/ckan/config/celery-supervisor.conf
```

Alternatively, you can run:

```
paster celeryd --config=/path/to/file.ini
```

Writing background tasks

These instructions should show you how to write an background task and how to call it from inside CKAN or another extension using celery.

Examples

Here are some existing real examples of writing CKAN tasks:

- <https://github.com/ckan/ckanext-archiver>
- <https://github.com/ckan/ckanext-qa>
- <https://github.com/ckan/ckanext-datastorer>

Setup

An entry point is required inside the `setup.py` for your extension, and so you should add something resembling the following that points to a function in a module. In this case the function is called `task_imports` in the `ckanext.NAME.celery_import` module:

```
entry_points = """
[ckan.celery_task]
tasks = ckanext.NAME.celery_import:task_imports
"""
```

The function, in this case `task_imports` should be a function that returns fully qualified module paths to modules that contain the defined task (see the next section). In this case we will put all of our tasks in a file called `tasks.py` and so `task_imports` should be in a file called `ckanext/NAME/celery_import.py`:

```
def task_imports():
    return ['ckanext.NAME.tasks']
```

This returns an iterable of all of the places to look to find tasks, in this example we are only putting them in one place.

Implementing the tasks

The most straightforward way of defining tasks in our `tasks.py` module, is to use the decorators provided by celery. These decorators make it easy to just define a function and then give it a name and make it accessible to celery. Make sure you import celery from `ckan.lib.celery_app`:

```
from ckan.lib.celery_app import celery
```

Implement your function, specifying the arguments you wish it to take. For our sample we will use a simple echo task that will print out its argument to the console:

```
def echo( message ):
    print message
```

Next it is important to decorate your function with the celery task decorator. You should give the task a name, which is used later on when calling the task:

```
@celery.task(name = "NAME.echofunction")
def echo( message ):
    print message
```

That's it, your function is ready to be run asynchronously outside of the main execution of the CKAN app. Next you should make sure you run `python setup.py develop` in your extensions folder and then go to your CKAN installation folder (normally `pyenv/src/ckan/`) to run the following command:

```
paster celeryd
```

Once you have done this your task name `NAME.echofunction` should appear in the list of tasks loaded. If it is there then you are all set and ready to go. If not then you should try the following to try and resolve the problem:

1. Make sure the entry point is defined correctly in your `setup.py` and that you have executed `python setup.py develop`
2. Check that your `task_imports` function returns an iterable with valid module names in
3. Ensure that the decorator marks the functions (if there is more than one decorator, make sure the `celery.task` is the first one - which means it will execute last).
4. If none of the above helps, go into `#ckan` on `irc.freenode.net` where there should be people who can help you resolve your issue.

Calling the task

Now that the task is defined, and has been loaded by celery it is ready to be called. To call a background task you need to know only the name of the task, and the arguments that it expects as well as providing it a task id.:

```
import uuid
from ckan.lib.celery_app import celery
celery.send_task("NAME.echofunction", args=["Hello World"], task_id=str(uuid.uuid4()))
```

After executing this code you should see the message printed in the console where you ran `paster celeryd`.

Retrying on errors

Should your task fail to complete because of a transient error, it is possible to ask celery to retry the task, after some period of time. The default wait before retrying is three minutes, but you can optionally specify this in the call to retry via the `countdown` parameter, and you can also specify the exception that triggered the failure. For our example the call to retry would look like the following - note that it calls the function name, not the task name given in the decorator:

```
try:
    ... some work that may fail, http request?
except Exception, e:
    # Retry again in 2 minutes
    echo.retry(args=(message), exc=e, countdown=120, max_retries=10)
```

If you don't want to wait a period of time you can use the `eta` datetime parameter to specify an explicit time to run the task (i.e. 9AM tomorrow)

Email notifications

CKAN can send email notifications to users, for example when a user has new activities on her dashboard. Once email notifications have been enabled by a site admin, each user of a CKAN site can turn email notifications on or off for herself by logging in and editing her user preferences. To enable email notifications for a CKAN site, a sysadmin must:

1. Setup a cron job or other scheduled job on a server to call CKAN's `send_email_notifications` API action at regular intervals (e.g. hourly) and send any pending email notifications to users.

On most UNIX systems you can setup a cron job by running `crontab -e` in a shell to edit your crontab file, and adding a line to the file to specify the new job. For more information run `man crontab` in a shell.

CKAN API actions can be called via the `paster post` command, which simulates an HTTP-request. For example, here is a crontab line to send out CKAN email notifications hourly:

```
@hourly echo '{}' | /usr/lib/ckan/bin/paster --plugin=ckan post -c /etc/ckan/production.ini /api
```

The `@hourly` can be replaced with `@daily`, `@weekly` or `@monthly`.

Warning: CKAN will not send email notifications for events older than the time period specified by the `ckan.email_notifications_since` config setting (default: 2 days), so your cron job should run more frequently than this. `@hourly` and `@daily` are good choices.

Note: Since `send_email_notifications` is an API action, it can be called from a machine other than the server on which CKAN is running, simply by POSTing an HTTP request to the CKAN API (you must be a sysadmin to call this particular API action). See [API guide](#).

2. CKAN will not send out any email notifications, nor show the email notifications preference to users, unless the `ckan.activity_streams_email_notifications` option is set to `True`, so put this line in the `[app:main]` section of your CKAN config file:

```
ckan.activity_streams_email_notifications = True
```

3. Make sure that `ckan.site_url` is set correctly in the `[app:main]` section of your CKAN configuration file. This is used to generate links in the bodies of the notification emails. For example:

```
ckan.site_url = http://publicdata.eu
```

4. Make sure that *smtp.mail_from* is set correctly in the `[app:main]` section of your CKAN configuration file. This is the email address that CKAN's email notifications will appear to come from. For example:

```
smtp.mail_from = mailman@publicdata.eu
```

This is combined with your *ckan.site_title* to form the `From:` header of the email that are sent, for example:

```
From: PublicData.eu <mailmain@publicdata.eu>
```

5. If you do not have an SMTP server running locally on the machine that hosts your CKAN instance, you can change the *Email Settings* to send email via an external SMTP server. For example, these settings in the `[app:main]` section of your configuration file will send emails using a gmail account (not recommended for production websites!):

```
smtp.server = smtp.gmail.com:587
smtp.starttls = True
smtp.user = your_username@gmail.com
smtp.password = your_gmail_password
smtp.mail_from = your_username@gmail.com
```

6. For the new configuration to take effect you need to restart the web server. For example if your are using Apache on Ubuntu, run this command in a shell:

```
sudo service apache2 reload
```

Page View Tracking

CKAN can track visits to pages of your site and use this tracking data to:

- Sort datasets by popularity
- Highlight popular datasets and resources
- Show view counts next to datasets and resources
- Show a list of the most popular datasets
- Export page-view data to a CSV file

See also:

ckanext-googleanalytics A CKAN extension that integrates Google Analytics into CKAN.

Enabling Page View Tracking

To enable page view tracking:

1. Set *ckan.tracking_enabled* to true in the `[app:main]` section of your CKAN configuration file (e.g. `development.ini` or `production.ini`):

```
[app:main]
ckan.tracking_enabled = true
```

Save the file and restart your web server. CKAN will now record raw page view tracking data in your CKAN database as pages are viewed.

2. Setup a cron job to update the tracking summary data.

For operations based on the tracking data CKAN uses a summarised version of the data, not the raw tracking data that is recorded “live” as page views happen. The `paster tracking update` and `paster search-index rebuild` commands need to be run periodically to update this tracking summary data.

You can setup a cron job to run these commands. On most UNIX systems you can setup a cron job by running `crontab -e` in a shell to edit your crontab file, and adding a line to the file to specify the new job. For more information run `man crontab` in a shell. For example, here is a crontab line to update the tracking data and rebuild the search index hourly:

```
@hourly /usr/lib/ckan/bin/paster --plugin=ckan tracking update -c /etc/ckan/production.ini && /u
```

Replace `/usr/lib/ckan/bin/` with the path to the `bin` directory of the virtualenv that you’ve installed CKAN into, and replace `/etc/ckan/production.ini` with the path to your CKAN configuration file.

The `@hourly` can be replaced with `@daily`, `@weekly` or `@monthly`.

Retrieving Tracking Data

Tracking summary data for datasets and resources is available in the dataset and resource dictionaries returned by, for example, the `package_show()` API:

```
"tracking_summary": {
  "recent": 5,
  "total": 15
},
```

This can be used, for example, by custom templates to show the number of views next to datasets and resources. A dataset or resource’s `recent` count is its number of views in the last 14 days, the `total` count is all of its tracked views (including recent ones).

You can also export tracking data for all datasets to a CSV file using the `paster tracking export` command. For details, run `paster tracking -h`.

Note: Repeatedly visiting the same page will not increase the page’s view count! Page view counting is limited to one view per user per page per day.

Sorting Datasets by Popularity

Once you’ve enabled page view tracking on your CKAN site, you can view datasets most-popular-first by selecting `Popular` from the `Order by:` dropdown on the dataset search page:

The screenshot shows the CKAN Datasets page. On the left, there are filters for Groups (Dave's books (2), Roger's books (1), Show More Groups), Tags (russian (2), Flexible ア (2), tolstoy (1), Show More Tags), and Formats (plain text (1), Clear All). The main content area shows a search bar, a dropdown for 'Order by' (Relevance, Relevance, Name Ascending, Name Descending, Last Modified, Popular), and two datasets: 'A Novel By Tolstoy' and 'A Wonderful Story'. The 'A Novel By Tolstoy' dataset has a 'Popular' badge and a tooltip showing '20 recent views'.

The datasets are sorted by their number of recent views.

You can retrieve datasets most-popular-first from the [CKAN API](#) by passing `'sort': 'views_recent desc'` to the `package_search()` action. This could be used, for example, by a custom template to show a list of the most popular datasets on the site's front page.

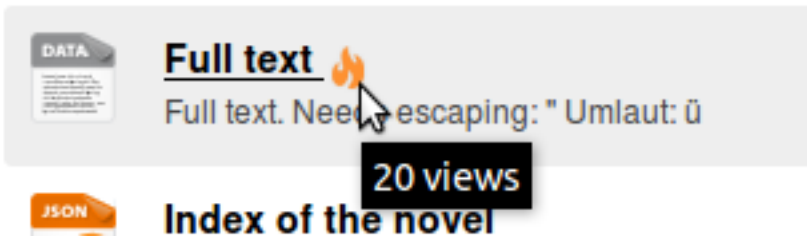
Tip: You can also sort datasets by total views rather than recent views. Pass `'sort': 'views_total desc'` to the `package_search()` API, or use the URL `/dataset?q=&sort=views_total+desc` in the web interface.

Highlighting Popular Datasets and Resources

Once you've enabled page view tracking on your CKAN site, popular datasets and resources (those with more than 10 views) will be highlighted with a "popular" badge and a tooltip showing the number of views:

The screenshot shows a dataset titled 'A Novel By Tolstoy' with a 'Popular' badge (a flame icon) and a tooltip showing '20 recent views'. Below the title, there is a description: 'Some test notes A 3rd level heading Some bolded text. Some italicized text. Foreign characters. ü with umlaut ü 66-style quote " thumb Needs...'. At the bottom, there are two buttons: 'plain text' and 'JSON'.

Data and Resources



Multilingual Extension

For translating CKAN's web interface see [Translating CKAN](#). In addition to user interface internationalization, a CKAN administrator can also enter translations into CKAN's database for terms that may appear in the contents of datasets, groups or tags created by users. When a user is viewing the CKAN site, if the translation terms database contains a translation in the user's language for the name or description of a dataset or resource, the name of a tag or group, etc. then the translated term will be shown to the user in place of the original.

Setup and Configuration

By default term translations are disabled. To enable them, you have to specify the multilingual plugins using the `ckan.plugins` setting in your CKAN configuration file, for example:

```
# List the names of CKAN extensions to activate.
ckan.plugins = multilingual_dataset multilingual_group multilingual_tag
```

Of course, you won't see any terms getting translated until you load some term translations into the database. You can do this using the `term_translation_update` and `term_translation_update_many` actions of the CKAN API, See [API guide](#) for more details.

Loading Test Translations

If you want to quickly test the term translation feature without having to provide your own translations, you can load CKAN's test translations into the database by running this command from your shell:

```
paster --plugin=ckan create-test-data translations
```

See [Command Line Interface](#) for more details.

Testing The Multilingual Extension

If you have a source installation of CKAN you can test the multilingual extension by running the tests located in `ckanext/multilingual/tests`. You must first install the packages needed for running CKAN tests into your virtual environment, and then run this command from your shell:

```
nosetests --ckan ckanext/multilingual/tests
```

See [Testing CKAN](#) for more information.

Stats Extension

CKAN's stats extension analyzes your CKAN database and displays several tables and graphs with statistics about your site, including:

- Total number of datasets
- Dataset revisions per week
- Top-rated datasets
- Most-edited Datasets
- Largest groups
- Top tags
- Users owning most datasets

See also:

CKAN's *built-in page view tracking feature*, which tracks visits to pages.

See also:

ckanext-googleanalytics A CKAN extension that integrates Google Analytics into CKAN.

Enabling the Stats Extension

To enable the stats extensions add `stats` to the *ckan.plugins* option in your CKAN config file, for example:

```
ckan.plugins = stats
```

If you also set the *ckanext.stats.cache_enabled* option to `true`, CKAN will cache the stats for one day instead of calculating them each time a user visits the stats page.

Viewing the Statistics

To view the statistics reported by the stats extension, visit the `/stats` page, for example: <http://demo.ckan.org/stats>

Configuration Options

The functionality and features of CKAN can be modified using many different configuration options. These are generally set in the [CKAN configuration file](#), but some of them can also be set via Environment variables or at *runtime*.

Environment variables

Some of the CKAN configuration options can be defined as Environment variables on the server operating system.

These are generally low-level critical settings needed when setting up the application, like the database connection, the Solr server URL, etc. Sometimes it can be useful to define them as environment variables to automate and orchestrate deployments without having to first modify the configuration file.

These options are only read at startup time to update the `config` object used by CKAN, but they won't be accessed any more during the lifetime of the application.

CKAN environment variables names match the options in the configuration file, but they are always uppercase and prefixed with `CKAN_` (this prefix is added even if the corresponding option in the ini file does not have it), and replacing dots with underscores.

This is the list of currently supported environment variables, please refer to the entries in the configuration file section below for more details about each one:

```
CONFIG_FROM_ENV_VARS = {
    'sqlalchemy.url': 'CKAN_SQLALCHEMY_URL',
    'ckan.datastore.write_url': 'CKAN_DATASTORE_WRITE_URL',
    'ckan.datastore.read_url': 'CKAN_DATASTORE_READ_URL',
    'solr_url': 'CKAN_SOLR_URL',
    'ckan.site_id': 'CKAN_SITE_ID',
    'ckan.site_url': 'CKAN_SITE_URL',
    'ckan.storage_path': 'CKAN_STORAGE_PATH',
    'ckan.datapusher.url': 'CKAN_DATAPUSHER_URL',
    'smtp.server': 'CKAN_SMTP_SERVER',
    'smtp.starttls': 'CKAN_SMTP_STARTTLS',
    'smtp.user': 'CKAN_SMTP_USER',
    'smtp.password': 'CKAN_SMTP_PASSWORD',
    'smtp.mail_from': 'CKAN_SMTP_MAIL_FROM'
}
```

Updating configuration options during runtime

CKAN configuration options are generally defined before starting the web application (either in the configuration file or via Environment variables).

A limited number of configuration options can also be edited during runtime. This can be done on the [administration interface](#) or using the `config_option_update()` API action. Only *sysadmins* can edit these runtime-editable configuration options. Changes made to these configuration options will be stored on the database and persisted when the server is restarted.

Extensions can add (or remove) configuration options to the ones that can be edited at runtime. For more details on how to this check [Making configuration options runtime-editable](#).

CKAN configuration file

By default, the configuration file is located at `/etc/ckan/default/development.ini` or `/etc/ckan/default/production.ini`. This section documents all of the config file settings, for reference.

Note: After editing your config file, you need to restart your webserver for the changes to take effect.

Note: Unless otherwise noted, all configuration options should be set inside the `[app:main]` section of the config file (i.e. after the `[app:main]` line):

```
[DEFAULT]

...

[server:main]
use = egg:Paste#http
host = 0.0.0.0
port = 5000
```

```
# This setting will not work, because it's outside of [app:main].
ckan.site_logo = /images/masaq.png

[app:main]
# This setting will work.
ckan.plugins = stats text_view recline_view
```

If the same option is set more than once in your config file, the last setting given in the file will override the others.

General Settings

debug

Example:

```
debug = False
```

Default value: False

This enables Pylons' interactive debugging tool, makes Fanstatic serve unminified JS and CSS files, and enables CKAN templates' debugging features.

Warning: This option should be set to `False` for a public site. With debug mode enabled, a visitor to your site could execute malicious commands.

Repoze.who Settings

who.timeout

Example:

```
who.timeout = 3600
```

Default value: None

This defines how long (in seconds) until a user is logged out after a period of inactivity. If the setting isn't defined, the session doesn't expire. Not active by default.

who.httponly

Default value: True

This determines whether the `HttpOnly` flag will be set on the repoze.who authorization cookie. The default in the absence of the setting is `True`. For enhanced security it is recommended to use the `HttpOnly` flag and not set this to `False`, unless you have a good reason for doing so.

who.secure

Example:


```
who.secure = True
```

Default value: False

This determines whether the secure flag will be set for the repoze.who authorization cookie. If `True`, the cookie will be sent over HTTPS. The default in the absence of the setting is `False`.

Database Settings

`sqlalchemy.url`

Example:

```
sqlalchemy.url = postgres://tester:pass@localhost/ckantest3
```

This defines the database that CKAN is to use. The format is:

```
sqlalchemy.url = postgres://USERNAME:PASSWORD@HOST/DBNAME
```

`ckan.datastore.write_url`

Example:

```
ckan.datastore.write_url = postgresql://ckanuser:pass@localhost/datastore
```

The database connection to use for writing to the datastore (this can be ignored if you're not using the *DataStore extension*). Note that the database used should not be the same as the normal CKAN database. The format is the same as in *sqlalchemy.url*.

`ckan.datastore.read_url`

Example:

```
ckan.datastore.read_url = postgresql://readonlyuser:pass@localhost/datastore
```

The database connection to use for reading from the datastore (this can be ignored if you're not using the *DataStore extension*). The database used must be the same used in *ckan.datastore.write_url*, but the user should be one with read permissions only. The format is the same as in *sqlalchemy.url*.

`ckan.datastore.sqlalchemy.*`

Example:

```
ckan.datastore.sqlalchemy.pool_size=10
ckan.datastore.sqlalchemy.max_overflow=20
```

Custom sqlalchemy config parameters used to establish the DataStore database connection.

To get the list of all the available properties check the [SQLAlchemy documentation](#)

`ckan.datastore.default_fts_lang`

Example:

```
ckan.datastore.default_fts_lang = english
```

Default value: `english`

This can be ignored if you're not using the *DataStore extension*.

The default language used when creating full-text search indexes and querying them. It can be overwritten by the user by passing the “lang” parameter to “datastore_search” and “datastore_create”.

`ckan.datastore.default_fts_index_method`

Example:

```
ckan.datastore.default_fts_index_method = gist
```

Default value: `gist`

This can be ignored if you're not using the *DataStore extension*.

The default method used when creating full-text search indexes. Currently it can be “gin” or “gist”. Refer to PostgreSQL's documentation to understand the characteristics of each one and pick the best for your instance.

Site Settings

`ckan.site_url`

Example:

```
ckan.site_url = http://scotdata.ckan.net
```

Default value: (an explicit value is mandatory)

The URL of your CKAN site. Many CKAN features that need an absolute URL to your site use this setting.

Important: It is mandatory to complete this setting

Warning: This setting should not have a trailing / on the end.

`apikey_header_name`

Example:

```
apikey_header_name = API-KEY
```

Default value: `X-CKAN-API-Key & Authorization`

This allows another http header to be used to provide the CKAN API key. This is useful if network infrastructure blocks the Authorization header and X-CKAN-API-Key is not suitable.

`ckan.cache_expires`

Example:

```
ckan.cache_expires = 2592000
```

Default value: 0

This sets Cache-Control header's max-age value.

`ckan.page_cache_enabled`

Example:

```
ckan.page_cache_enabled = True
```

Default value: False

This enables CKAN's built-in page caching.

Warning: Page caching is an experimental feature.

`ckan.cache_enabled`

Example:

```
ckan.cache_enabled = True
```

Default value: None

Controls if we're caching CKAN's static files, if it's serving them.

`ckan.use_pylons_response_cleanup_middleware`

Example:

```
ckan.use_pylons_response_cleanup_middleware = true
```

Default value: true

This enables middleware that clears the response string after it has been sent. This helps CKAN's memory management if CKAN repeatedly serves very large requests.

`ckan.static_max_age`

Example:

```
ckan.static_max_age = 2592000
```

Default value: 3600

Controls CKAN static files' cache max age, if we're serving and caching them.

ckan.tracking_enabled

Example:

```
ckan.tracking_enabled = True
```

Default value: False

This controls if CKAN will track the site usage. For more info, read [Page View Tracking](#).

Authorization Settings

More information about how authorization works in CKAN can be found the [Organizations and authorization](#) section.

ckan.auth.anon_create_dataset

Example:

```
ckan.auth.anon_create_dataset = False
```

Default value: False

Allow users to create datasets without registering and logging in.

ckan.auth.create_unowned_dataset

Example:

```
ckan.auth.create_unowned_dataset = False
```

Default value: True

Allow the creation of datasets not owned by any organization.

ckan.auth.create_dataset_if_not_in_organization

Example:

```
ckan.auth.create_dataset_if_not_in_organization = False
```

Default value: True

Allow users who are not members of any organization to create datasets, default: true. `create_unowned_dataset` must also be True, otherwise setting `create_dataset_if_not_in_organization` to True is meaningless.

ckan.auth.user_create_groups

Example:

```
ckan.auth.user_create_groups = False
```

Default value: True

Allow users to create groups.

ckan.auth.user_create_organizations

Example:

```
ckan.auth.user_create_organizations = False
```

Default value: True

Allow users to create organizations.

ckan.auth.user_delete_groups

Example:

```
ckan.auth.user_delete_groups = False
```

Default value: True

Allow users to delete groups.

ckan.auth.user_delete_organizations

Example:

```
ckan.auth.user_delete_organizations = False
```

Default value: True

Allow users to delete organizations.

ckan.auth.create_user_via_api

Example:

```
ckan.auth.create_user_via_api = False
```

Default value: False

Allow new user accounts to be created via the API.

ckan.auth.create_user_via_web

Example:

```
ckan.auth.create_user_via_web = True
```

Default value: True

Allow new user accounts to be created via the Web.

ckan.auth.roles_that_cascade_to_sub_groups

Example:

```
ckan.auth.roles_that_cascade_to_sub_groups = admin editor
```

Default value: admin

Makes role permissions apply to all the groups down the hierarchy from the groups that the role is applied to.

e.g. a particular user has the ‘admin’ role for group ‘Department of Health’. If you set the value of this option to ‘admin’ then the user will automatically have the same admin permissions for the child groups of ‘Department of Health’ such as ‘Cancer Research’ (and its children too and so on).

Search Settings

ckan.site_id

Example:

```
ckan.site_id = my_ckan_instance
```

CKAN uses Solr to index and search packages. The search index is linked to the value of the `ckan.site_id`, so if you have more than one CKAN instance using the same `solr_url`, they will each have a separate search index as long as their `ckan.site_id` values are different. If you are only running a single CKAN instance then this can be ignored.

Note, if you change this value, you need to rebuild the search index.

ckan.simple_search

Example:

```
ckan.simple_search = true
```

Default value: false

Switching this on tells CKAN search functionality to just query the database, (rather than using Solr). In this setup, search is crude and limited, e.g. no full-text search, no faceting, etc. However, this might be very useful for getting up and running quickly with CKAN.

solr_url

Example:

```
solr_url = http://solr.okfn.org:8983/solr/ckan-schema-2.0
```

Default value: `http://127.0.0.1:8983/solr`

This configures the Solr server used for search. The Solr schema found at that URL must be one of the ones in `ckan/config/solr` (generally the most recent one). A check of the schema version number occurs when CKAN starts.

Optionally, `solr_user` and `solr_password` can also be configured to specify HTTP Basic authentication details for all Solr requests.

Note: If you change this value, you need to rebuild the search index.

ckan.search.automatic_indexing

Example:

```
ckan.search.automatic_indexing = true
```

Default value: `true`

Make all changes immediately available via the search after editing or creating a dataset. Default is true. If for some reason you need the indexing to occur asynchronously, set this option to false.

Note: This is equivalent to explicitly load the `synchronous_search` plugin.

ckan.search.solr_commit

Example:

```
ckan.search.solr_commit = false
```

Default value: `true`

Make ckan commit changes solr after every dataset update change. Turn this to false if on solr 4.0 and you have automatic (soft)commits enabled to improve dataset update/create speed (however there may be a slight delay before dataset gets seen in results).

ckan.search.show_all_types

Example:

```
ckan.search.show_all_types = true
```

Default value: `false`

Controls whether the default search page (`/dataset`) should show only standard datasets or also custom dataset types.

search.facets.limit

Example:

```
search.facets.limit = 100
```

Default value: `50`

Sets the default number of searched facets returned in a query.

search.facets.default

Example:

```
search.facets.default = 10
```

Default number of facets shown in search results. Default 10.

`ckan.extra_resource_fields`

Example:

```
ckan.extra_resource_fields = alt_url
```

Default value: None

List of the extra resource fields that would be used when searching.

CORS Settings

Cross-Origin Resource Sharing (CORS) can be enabled and controlled with the following settings:

`ckan.cors.origin_allow_all`

Example:

```
ckan.cors.origin_allow_all = True
```

This setting must be present to enable CORS. If True, all origins will be allowed (the response header Access-Control-Allow-Origin is set to '*'). If False, only origins from the `ckan.cors.origin_whitelist` setting will be allowed.

`ckan.cors.origin_whitelist`

Example:

```
ckan.cors.origin_whitelist = http://www.myremotedomain1.com http://myremotedomain1.com
```

A space separated list of allowable origins. This setting is used when `ckan.cors.origin_allow_all = False`.

Plugins Settings

`ckan.plugins`

Example:

```
ckan.plugins = disqus datapreview googleanalytics follower
```

Default value: stats text_view recline_view

Specify which CKAN plugins are to be enabled.

Warning: If you specify a plugin but have not installed the code, CKAN will not start.

Format as a space-separated list of the plugin names. The plugin name is the key in the `[ckan.plugins]` section of the extension's `setup.py`. For more information on plugins and extensions, see [Extending guide](#).

Note: The order of the plugin names in the configuration file influences the order that CKAN will load the plugins in. As long as each plugin class is implemented in a separate Python module (i.e. in a separate Python source code file), the plugins will be loaded in the order given in the configuration file.

When multiple plugins are implemented in the same Python module, CKAN will process the plugins in the order that they're given in the config file, but as soon as it reaches one plugin from a given Python module, CKAN will load all plugins from that Python module, in the order that the plugin classes are defined in the module.

For simplicity, we recommend implementing each plugin class in its own Python module.

Plugin loading order can be important, for example for plugins that add custom template files: templates found in template directories added earlier will override templates in template directories added later.

Todo

Fix CKAN's plugin loading order to simply load all plugins in the order they're given in the config file, regardless of which Python modules they're implemented in.

`ckan.datastore.enabled`

Example:

```
ckan.datastore.enabled = True
```

Default value: False

Controls if the Data API link will appear in Dataset's Resource page.

Note: This setting only applies to the legacy templates.

`ckanext.stats.cache_enabled`

Example:

```
ckanext.stats.cache_enabled = True
```

Default value: True

This controls if we'll use the 1 day cache for stats.

`ckan.resource_proxy.max_file_size`

Example:

```
ckan.resource_proxy.max_file_size = 1 * 1024 * 1024
```

Default value: 1 * 1024 * 1024 (1 MB)

This sets the upper file size limit for in-line previews. Increasing the value allows CKAN to preview larger files (e.g. PDFs) in-line; however, a higher value might cause time-outs, or unresponsive browsers for CKAN users with lower bandwidth. If left commented out, CKAN will default to 1 MB.

Front-End Settings

`ckan.site_title`

Example:

```
ckan.site_title = Open Data Scotland
```

Default value: CKAN

This sets the name of the site, as displayed in the CKAN web interface.

ckan.site_description

Example:

```
ckan.site_description = The easy way to get, use and share data
```

Default value: (none)

This is for a description, or tag line for the site, as displayed in the header of the CKAN web interface.

ckan.site_intro_text

Example:

```
ckan.site_intro_text = Nice introductory paragraph about CKAN or the site in general.
```

Default value: (none)

This is for an introductory text used in the default template's index page.

ckan.site_logo

Example:

```
ckan.site_logo = /images/ckan_logo_fullname_long.png
```

Default value: (none)

This sets the logo used in the title bar.

ckan.site_about

Example:

```
ckan.site_about = A _community-driven_ catalogue of _open data_ for the Greenfield area.
```

Default value:

```
<p>CKAN is the world's leading open-source data portal platform.</p>
```

```
<p>CKAN is a complete out-of-the-box software solution that makes data accessible and usable - by providing tools to streamline publishing, sharing, finding and using data (including storage of data and provision of robust data APIs). CKAN is aimed at data publishers (national and regional governments, companies and organizations) wanting to make their data open and available.</p>
```

```
<p>CKAN is used by governments and user groups worldwide and powers a variety of official and community data portals including portals for local, national and international government, such as the UK's <a href="http://data.gov.uk">data.gov.uk</a> and the European Union's <a href="http://publicdata.eu/">publicdata.eu</a>,</p>
```

the Brazilian `dados.gov.br`, Dutch and Netherland government portals, as well as city and municipal sites in the US, UK, Argentina, Finland and elsewhere.

```
<p>CKAN: <a href="http://ckan.org/">http://ckan.org/</a><br />
CKAN Tour: <a href="http://ckan.org/tour/">http://ckan.org/tour/</a><br />
Features overview: <a href="http://ckan.org/features/">http://ckan.org/features/</a></p>
```

Format tips:

- multiline strings can be used by indenting following lines
- the format is Markdown

Note: Whilst the default text is translated into many languages (switchable in the page footer), the text in this configuration option will not be translatable. For this reason, it's better to overload the snippet in `home/snippets/about_text.html`. For more information, see [Theming guide](#).

`ckan.main_css`

Example:

```
ckan.main_css = /base/css/my-custom.css
```

Default value: `/base/css/main.css`

With this option, instead of using the default *main.css*, you can use your own.

`ckan.favicon`

Example:

```
ckan.favicon = http://okfn.org/wp-content/themes/okfn-master-wordpress-theme/images/favicon.ico
```

Default value: `/images/icons/ckan.ico`

This sets the site's *favicon*. This icon is usually displayed by the browser in the tab heading and bookmark.

`ckan.legacy_templates`

Example:

```
ckan.legacy_templates = True
```

Default value: `False`

This controls if the legacy genshi templates are used.

Note: This is only for legacy code, and shouldn't be used anymore.

`ckan.datasets_per_page`

Example:

```
ckan.datasets_per_page = 10
```

Default value: 20

This controls the pagination of the dataset search results page. This is the maximum number of datasets viewed per page of results.

package_hide_extras

Example:

```
package_hide_extras = my_private_field other_field
```

Default value: (empty)

This sets a space-separated list of extra field key values which will not be shown on the dataset read page.

Warning: While this is useful to e.g. create internal notes, it is not a security measure. The keys will still be available via the API and in revision diffs.

ckan.dataset.show_apps_ideas

ckan.dataset.show_apps_ideas:

```
ckan.dataset.show_apps_ideas = false
```

Default value: true

When set to false, or no, this setting will hide the ‘Apps, Ideas, etc’ tab on the package read page. If the value is not set, or is set to true or yes, then the tab will shown.

Note: This only applies to the legacy Genshi-based templates

ckan.dumps_url

If there is a page which allows you to download a dump of the entire catalogue then specify the URL here, so that it can be advertised in the web interface. For example:

```
ckan.dumps_url = http://ckan.net/dump/
```

For more information on using dumpfiles, see [db: Manage databases](#).

ckan.dumps_format

If there is a page which allows you to download a dump of the entire catalogue then specify the format here, so that it can be advertised in the web interface. `dumps_format` is just a string for display. Example:

```
ckan.dumps_format = CSV/JSON
```

ckan.recaptcha.version

The version of Recaptcha to use, for example:

```
ckan.recaptcha.version = 1
```

Default Value: 1

Valid options: 1, 2

ckan.recaptcha.publickey

The public key for your Recaptcha account, for example:

```
ckan.recaptcha.publickey = 6Lc...-KLc
```

To get a Recaptcha account, sign up at: <http://www.google.com/recaptcha>

ckan.recaptcha.privatekey

The private key for your Recaptcha account, for example:

```
ckan.recaptcha.privatekey = 6Lc...-jP
```

Setting both *ckan.recaptcha.publickey* and *ckan.recaptcha.privatekey* adds captcha to the user registration form. This has been effective at preventing bots registering users and creating spam packages.

ckan.featured_groups

Example:

```
ckan.featured_groups = group_one
```

Default Value: (empty)

Defines a list of group names or group ids. This setting is used to display a group and datasets on the home page in the default templates (1 group and 2 datasets are displayed).

ckan.featured_orgs

Example:

```
ckan.featured_orgs = org_one
```

Default Value: (empty)

Defines a list of organization names or ids. This setting is used to display an organization and datasets on the home page in the default templates (1 group and 2 datasets are displayed).

ckan.gravatar_default

Example:

```
ckan.gravatar_default = monsterid
```

Default value: `identicon`

This controls the default gravatar avatar, in case the user has none.

ckan.debug_supress_header

Example:

```
ckan.debug_supress_header = False
```

Default value: `False`

This config if the debug information showing the controller and action receiving the request being is shown in the header.

Note: This info only shows if debug is set to `True`.

Resource Views Settings

ckan.views.default_views

Example:

```
ckan.views.default_views = image_view webpage_view recline_grid_view
```

Default value: `image_view recline_view`

Defines the resource views that should be created by default when creating or updating a dataset. From this list only the views that are relevant to a particular resource format will be created. This is determined by each individual view.

If not present (or commented), the default value is used. If left empty, no default views are created.

Note: You must have the relevant view plugins loaded on the `ckan.plugins` setting to be able to create the default views, eg:

```
ckan.plugins = image_view webpage_view recline_grid_view ...
```

```
ckan.views.default_views = image_view webpage_view recline_grid_view
```

ckan.preview.json_formats

Example:

```
ckan.preview.json_formats = json
```

Default value: `json`

JSON based resource formats that will be rendered by the Text view plugin (`text_view`)

`ckan.preview.xml_formats`

Example:

```
ckan.preview.xml_formats = xml rdf rss
```

Default value: `xml rdf rdf+xml owl+xml atom rss`

XML based resource formats that will be rendered by the Text view plugin (`text_view`)

`ckan.preview.text_formats`

Example:

```
ckan.preview.text_formats = text plain
```

Default value: `text plain text/plain`

Plain text based resource formats that will be rendered by the Text view plugin (`text_view`)

Theming Settings

`ckan.template_head_end`

HTML content to be inserted just before `</head>` tag (e.g. extra stylesheet)

Example:

```
ckan.template_head_end = <link rel="stylesheet" href="http://mysite.org/css/custom.css" type="text/css">
```

You can also have multiline strings. Just indent following lines. e.g.:

```
ckan.template_head_end =
<link rel="stylesheet" href="/css/extra1.css" type="text/css">
<link rel="stylesheet" href="/css/extra2.css" type="text/css">
```

Note: This is only for legacy code, and shouldn't be used anymore.

`ckan.template_footer_end`

HTML content to be inserted just before `</body>` tag (e.g. Google Analytics code).

Note: you can have multiline strings (just indent following lines)

Example (showing insertion of Google Analytics code):

```
ckan.template_footer_end = <!-- Google Analytics -->
<script src='http://www.google-analytics.com/ga.js' type='text/javascript'></script>
<script type="text/javascript">
try {
var pageTracker = _gat._getTracker("XXXXXXXXXX");
pageTracker._setDomainName(".ckan.net");
pageTracker._trackPageview();
} catch(err) {}
```

```
</script>
<!-- /Google Analytics -->
```

Note: This is only for legacy code, and shouldn't be used anymore.

ckan.template_title_delimiter

Example:

```
ckan.template_title_delimiter = |
```

Default value: –

This sets the delimiter between the site's subtitle (if there's one) and its title, in HTML's <title>.

extra_template_paths

Example:

```
extra_template_paths = /home/okfn/brazil_ckan_config/templates
```

To customise the display of CKAN you can supply replacements for the Genshi template files. Use this option to specify where CKAN should look for additional templates, before reverting to the `ckan/templates` folder. You can supply more than one folder, separating the paths with a comma (,).

For more information on theming, see [Theming guide](#).

extra_public_paths

Example:

```
extra_public_paths = /home/okfn/brazil_ckan_config/public
```

To customise the display of CKAN you can supply replacements for static files such as HTML, CSS, script and PNG files. Use this option to specify where CKAN should look for additional files, before reverting to the `ckan/public` folder. You can supply more than one folder, separating the paths with a comma (,).

For more information on theming, see [Theming guide](#).

Storage Settings

ckan.storage_path

Example:: `ckan.storage_path = /var/lib/ckan`

Default value: None

This defines the location of where CKAN will store all uploaded data.

ckan.max_resource_size

Example:: `ckan.max_resource_size = 100`

Default value: 10

The maximum in megabytes a resources upload can be.

ckan.max_image_size

Example:: `ckan.max_image_size = 10`

Default value: 2

The maximum in megabytes an image upload can be.

ofs.impl

Example:

```
ofs.impl = pairtree
```

Default value: None

Defines the storage backend used by CKAN: `pairtree` for local storage, `s3` for Amazon S3 Cloud Storage or `google` for Google Cloud Storage. Note that each of these must be accompanied by the relevant settings for each backend described below.

Deprecated, only available option is now `pairtree`. This must be used nonetheless if upgrading for CKAN 2.1 in order to keep access to your old `pairtree` files.

ofs.storage_dir

Example:

```
ofs.storage_dir = /data/uploads/
```

Default value: None

Only used with the local storage backend. Use this to specify where uploaded files should be stored, and also to turn on the handling of file storage. The folder should exist, and will automatically be turned into a valid `pairtree` repository if it is not already.

Deprecated, please use `ckan.storage_path`. This must be used nonetheless if upgrading for CKAN 2.1 in order to keep access to your old `pairtree` files.

DataPusher Settings**ckan.datapusher.formats**

Example:

```
ckan.datapusher.formats = csv xls
```

Default value: `csv xls.xlsx tsv application/csv application/vnd.ms-excel application/vnd.openxmlformats-officedocument.spreadsheetml.sheet`

File formats that will be pushed to the DataStore by the DataPusher. When adding or editing a resource which links to a file in one of these formats, the DataPusher will automatically try to import its contents to the DataStore.

ckan.datapusher.url

Example:

```
ckan.datapusher.url = http://127.0.0.1:8800/
```

DataPusher endpoint to use when enabling the `datapusher` extension. If you installed CKAN via *Installing CKAN from package*, the DataPusher was installed for you running on port 8800. If you want to manually install the DataPusher, follow the installation [instructions](#).

Activity Streams Settings

ckan.activity_streams_enabled

Example:

```
ckan.activity_streams_enabled = False
```

Default value: `True`

Turns on and off the activity streams used to track changes on datasets, groups, users, etc

ckan.activity_streams_email_notifications

Example:

```
ckan.activity_streams_email_notifications = False
```

Default value: `False`

Turns on and off the activity streams' email notifications. You'd also need to setup a cron job to send the emails. For more information, visit [Email notifications](#).

ckan.activity_list_limit

Example:

```
ckan.activity_list_limit = 31
```

Default value: `infinite`

This controls the number of activities to show in the Activity Stream. By default, it shows everything.

ckan.email_notifications_since

Example:

```
ckan.email_notifications_since = 2 days
```

Default value: `infinite`

Email notifications for events older than this time delta will not be sent. Accepted formats: ‘2 days’, ‘14 days’, ‘4:35:00’ (hours, minutes, seconds), ‘7 days, 3:23:34’, etc.

ckan.hide_activity_from_users

Example:

```
ckan.hide_activity_from_users = sysadmin
```

Hides activity from the specified users from activity stream. If unspecified, it’ll use *ckan.site_id* to hide activity by the site user. The site user is a sysadmin user on every ckan user with a username that’s equal to *ckan.site_id*. This user is used by ckan for performing actions from the command-line.

Feeds Settings

ckan.feeds.author_name

Example:

```
ckan.feeds.author_name = Michael Jackson
```

Default value: `(none)`

This controls the feed author’s name. If unspecified, it’ll use *ckan.site_id*.

ckan.feeds.author_link

Example:

```
ckan.feeds.author_link = http://okfn.org
```

Default value: `(none)`

This controls the feed author’s link. If unspecified, it’ll use *ckan.site_url*.

ckan.feeds.authority_name

Example:

```
ckan.feeds.authority_name = http://okfn.org
```

Default value: `(none)`

The domain name or email address of the default publisher of the feeds and elements. If unspecified, it’ll use *ckan.site_url*.

ckan.feeds.date

Example:

```
ckan.feeds.date = 2012-03-22
```

Default value: (none)

A string representing the default date on which the `authority_name` is owned by the publisher of the feed.

Internationalisation Settings

ckan.locale_default

Example:

```
ckan.locale_default = de
```

Default value: en (English)

Use this to specify the locale (language of the text) displayed in the CKAN Web UI. This requires a suitable *mo* file installed for the locale in the `ckan/i18n`. For more information on internationalization, see [Translating CKAN](#). If you don't specify a default locale, then it will default to the first locale offered, which is by default English (alter that with `ckan.locales_offered` and `ckan.locales_filtered_out`).

ckan.locales_offered

Example:

```
ckan.locales_offered = en de fr
```

Default value: (none)

By default, all locales found in the `ckan/i18n` directory will be offered to the user. To only offer a subset of these, list them under this option. The ordering of the locales is preserved when offered to the user.

ckan.locales_filtered_out

Example:

```
ckan.locales_filtered_out = pl ru
```

Default value: (none)

If you want to not offer particular locales to the user, then list them here to have them removed from the options.

ckan.locale_order

Example:

```
ckan.locale_order = fr de
```

Default value: (none)

If you want to specify the ordering of all or some of the locales as they are offered to the user, then specify them here in the required order. Any locales that are available but not specified in this option, will still be offered at the end of the list.

ckan.i18n_directory

Example:

```
ckan.i18n_directory = /opt/locales/i18n/
```

Default value: (none)

By default, the locales are searched for in the `ckan/i18n` directory. Use this option if you want to use another folder.

ckan.root_path

Example:

```
ckan.root_path = /my/custom/path/{{LANG}}/foo
```

Default value: (none)

By default, the URLs are formatted as `/some/url`, when using the default locale, or `/de/some/url` when using the “de” locale, for example. This lets you change this. You can use any path that you want, adding `{{LANG}}` where you want the locale code to go.

ckan.resource_formats

Example:: `ckan.resource_formats = /path/to/resource_formats`

Default value: `ckan/config/resource_formats.json`

The purpose of this file is to supply a thorough list of resource formats and to make sure the formats are normalized when saved to the database and presented.

The format of the file is a JSON object with following format:

```
[{"Format", "Description", "Mimetype", ["List of alternative representations"]}]
```

Please look in `ckan/config/resource_formats.json` for full details and as an example.

Form Settings

package_new_return_url

The URL to redirect the user to after they’ve submitted a new package form, example:

```
package_new_return_url = http://datadotgc.ca/new_dataset_complete?name=<NAME>
```

This is useful for integrating CKAN’s new dataset form into a third-party interface, see *Form Integration*.

The `<NAME>` string is replaced with the name of the dataset created.

package_edit_return_url

The URL to redirect the user to after they've submitted an edit package form, example:

```
package_edit_return_url = http://datadotgc.ca/dataset/<NAME>
```

This is useful for integrating CKAN's edit dataset form into a third-party interface, see [Form Integration](#).

The <NAME> string is replaced with the name of the dataset that was edited.

licenses_group_url

A url pointing to a JSON file containing a list of license objects. This list determines the licenses offered by the system to users, for example when creating or editing a dataset.

This is entirely optional - by default, the system will use an internal cached version of the CKAN list of licenses available from the <http://licenses.opendefinition.org/licenses/groups/ckan.json>.

More details about the license objects - including the license format and some example license lists - can be found at the [Open Licenses Service](#).

Examples:

```
licenses_group_url = file:///path/to/my/local/json-list-of-licenses.json  
licenses_group_url = http://licenses.opendefinition.org/licenses/groups/od.json
```

Email Settings

smtp.server

Example:

```
smtp.server = smtp.gmail.com:587
```

Default value: None

The SMTP server to connect to when sending emails with optional port.

smtp.starttls

Example:

```
smtp.starttls = True
```

Default value: None

Whether or not to use STARTTLS when connecting to the SMTP server.

smtp.user

Example:

```
smtp.user = your_username@gmail.com
```

Default value: None

The username used to authenticate with the SMTP server.

smtp.password

Example:

```
smtp.password = yourpass
```

Default value: None

The password used to authenticate with the SMTP server.

smtp.mail_from

Example:

```
smtp.mail_from = you@yourdomain.com
```

Default value: None

The email address that emails sent by CKAN will come from. Note that, if left blank, the SMTP server may insert its own.

email_to

Example:

```
email_to = you@yourdomain.com
```

Default value: None

This controls where the error messages will be sent to.

error_email_from

Example:

```
error_email_from = paste@localhost
```

Default value: None

This controls from which email the error messages will come from.

Multicore Solr setup

Solr can be set up to have multiple configurations and search indexes on the same machine. Each configuration is called a Solr *core*. Having multiple cores is useful when you want different applications or different versions of CKAN to share the same Solr instance, each application can have its own Solr core so each can use a different `schema.xml` file. This is necessary, for example, if you want two CKAN instances to share the same Solr server and those two instances are running different versions of CKAN that require different `schema.xml` files, or if the two instances have different Solr schema customizations.

Each Solr core in a multicore setup will have a different URL, for example:

```
http://localhost:8983/solr/ckan_default
http://localhost:8983/solr/some_other_site
```

This section will show you how to create a multicore Solr setup and create your first core. If you already have a multicore setup and now you've setup a second CKAN instance on the same machine and want to create a second Solr core for it, see [Creating another Solr core](#).

1. Create the file `/usr/share/solr/solr.xml`, with the following contents:

```
<solr persistent="true" sharedLib="lib">
  <cores adminPath="/admin/cores">
    <core name="ckan_default" instanceDir="ckan_default">
      <property name="dataDir" value="/var/lib/solr/data/ckan_default" />
    </core>
  </cores>
</solr>
```

This file lists the different Solr cores, in this example we have just a single core called `ckan_default`.

2. Create the data directory for your Solr core, run this command in a terminal:

```
sudo -u jetty mkdir /var/lib/solr/data/ckan_default
```

This is the directory where Solr will store the search index files for our core.

3. Create the directory `/etc/solr/ckan_default`, and move the `/etc/solr/conf` directory into it:

```
sudo mkdir /etc/solr/ckan_default
sudo mv /etc/solr/conf /etc/solr/ckan_default/
```

This directory holds the configuration files for your Solr core.

4. Replace the `/etc/solr/ckan_default/schema.xml` file with a symlink to CKAN's `schema.xml` file:

```
sudo mv /etc/solr/ckan_default/conf/schema.xml /etc/solr/ckan_default/conf/schema.xml.bak
sudo ln -s /usr/lib/ckan/default/src/ckan/ckan/config/solr/schema.xml /etc/solr/ckan_default/conf/schema.xml
```

5. Edit `/etc/solr/ckan_default/conf/solrconfig.xml` and change the `<dataDir>` tag to this:

```
<dataDir>${dataDir}</dataDir>
```

This configures our `ckan_default` core to use the data directory you specified for it in `solr.xml`.

6. Create the directory `/usr/share/solr/ckan_default` and put a symlink to the `conf` directory in it:

```
sudo mkdir /usr/share/solr/ckan_default
sudo ln -s /etc/solr/ckan_default/conf /usr/share/solr/ckan_default/conf
```

7. Restart Solr:

```
sudo service jetty restart
```

You should now see your newly created `ckan_default` core if you open http://localhost:8983/solr/ckan_default/admin/ in your web browser. You can click on the *schema* link on this page to check that the core is using the right schema (you should see `<schema name="ckan" version="2.0">` near the top of the `schema.xml` file). The <http://localhost:8983/solr/> page will list all of your configured Solr cores.

8. Finally, change the `solr_url` setting in your `/etc/ckan/default/development.ini` or `/etc/ckan/default/production.ini` file to point to your new Solr core, for example:


```
solr_url = http://127.0.0.1:8983/solr/ckan_default
```

If you have trouble when setting up Solr, see [Multicore Solr setup troubleshooting](#) below.

Creating another Solr core

In this example we'll assume that:

1. You've followed the instructions in [Multicore Solr setup](#) to create a multicore setup and create your first core for your first CKAN instance.
2. You've installed a second instance of CKAN in a second virtual environment at `/usr/lib/ckan/my-second-ckan-instance`, and now want to setup a second Solr core for it.

You can of course follow these instructions again to setup further Solr cores.

1. Add the core to `/usr/share/solr/solr.xml`. This file should now list two cores. For example:

```
<solr persistent='true' sharedLib='lib'>
  <cores adminPath='/admin/cores'>
    <core name='ckan_default' instanceDir='ckan_default'>
      <property name='dataDir' value='/var/lib/solr/data/ckan_default' />
    </core>
    <core name='my-second-solr-core' instanceDir='my-second-solr-core'>
      <property name='dataDir' value='/var/lib/solr/data/my-second-solr-core' />
    </core>
  </cores>
</solr>
```

2. Create the data directory for your new core:

```
sudo -u jetty mkdir /var/lib/solr/data/my-second-solr-core
```

3. Create the configuration directory for your new core, and copy the config from your first core into it:

```
sudo mkdir /etc/solr/my-second-solr-core
sudo cp -R /etc/solr/ckan_default/conf /etc/solr/my-second-solr-core/
```

4. Replace the `/etc/solr/my-second-solr-core/schema.xml` file with a symlink to the `schema.xml` file from your second CKAN instance:

```
sudo rm /etc/solr/my-second-solr-core/conf/schema.xml
sudo ln -s /usr/lib/ckan/my-second-ckan-instance/src/ckan/ckan/config/solr/schema.xml
```

5. Create the `/usr/share/solr/my-second-solr-core` directory and put a symlink to the `conf` directory in it:

```
sudo mkdir /usr/share/solr/my-second-solr-core
sudo ln -s /etc/solr/my-second-solr-core/conf /usr/share/solr/my-second-solr-core/conf
```

6. Restart Solr:

```
sudo service jetty restart
```

You should now see both your Solr cores when you open <http://localhost:8983/solr/> in your web browser.

7. Finally, change the `solr_url` setting in your `/etc/ckan/my-second-ckan-instance/development.ini` or `/etc/ckan/my-second-ckan-instance/production.ini` file to point to your new Solr core:

```
solr_url = http://127.0.0.1:8983/solr/my-second-solr-core
```

If you have trouble when setting up Solr, see [Multicore Solr setup troubleshooting](#).

Multicore Solr setup troubleshooting

See also:

Troubleshooting for single-core Solr setups Most of these tips also apply to multi-core setups.

No cores shown on Solr index page

If no cores are shown when you visit the Solr index page, and the admin interface returns a 404 error, check the web server error log (`/var/log/jetty/<date>.stderrout.log` if you're using Jetty, or `/var/log/tomcat6/catalina.<date>.log` for Tomcat). If you can find an error similar to this one:

```
WARNING: [iatiregistry.org] Solr index directory '/usr/share/solr/iatiregistry.org/data/index' doesn't exist
07-Dec-2011 18:06:33 org.apache.solr.common.SolrException log
SEVERE: java.lang.RuntimeException: Cannot create directory: /usr/share/solr/iatiregistry.org/data/index
[...]
```

Then `dataDir` is not properly configured. With our setup the data directory should be under `/var/lib/solr/data`. Make sure that you defined the correct `dataDir` in the `solr.xml` file and that in the `solrconfig.xml` file you have the following configuration option:

```
<dataDir>${dataDir}</dataDir>
```

API guide

This section documents CKAN's API, for developers who want to write code that interacts with CKAN sites and their data.

CKAN's **Action API** is a powerful, RPC-style API that exposes all of CKAN's core features to API clients. All of a CKAN website's core functionality (everything you can do with the web interface and more) can be used by external code that calls the CKAN API. For example, using the CKAN API your app can:

- Get JSON-formatted lists of a site's datasets, groups or other CKAN objects:

http://demo.ckan.org/api/3/action/package_list

http://demo.ckan.org/api/3/action/group_list

http://demo.ckan.org/api/3/action/tag_list

- Get a full JSON representation of a dataset, resource or other object:

http://demo.ckan.org/api/3/action/package_show?id=adur_district_spending

http://demo.ckan.org/api/3/action/tag_show?id=gold

http://demo.ckan.org/api/3/action/group_show?id=data-explorer

- Search for packages or resources matching a query:

http://demo.ckan.org/api/3/action/package_search?q=spending

http://demo.ckan.org/api/3/action/resource_search?query=name:District%20Names

- Create, update and delete datasets, resources and other objects

- Get an activity stream of recently changed datasets on a site:

http://demo.ckan.org/api/3/action/recently_changed_packages_activity_list

Note: CKAN's FileStore and DataStore have their own APIs, see:

- *FileStore and file uploads*
 - *DataStore extension*
-

Note: For documentation of CKAN's legacy API's, see *Legacy APIs*.

Legacy APIs

Warning: The legacy APIs documented in this section are provided for backwards-compatibility, but support for new CKAN features will not be added to these APIs.

API Versions

There are two versions of the legacy APIs. When the API returns a reference to an object, version 1 of the API will return the name of the object (e.g. "river-pollution"), whereas version 2 will return the ID of the object (e.g. "a3dd8f64-9078-4f04-845c-e3f047125028"). Tag objects are an exception, tag names are immutable so tags are always referred to with their name.

You can specify which version of the API to use in the URL. For example, opening this URL in your web browser will list demo.ckan.org's datasets using API version 1:

<http://demo.ckan.org/api/1/rest/dataset>

Opening this URL calls the same function using API version 2:

<http://demo.ckan.org/api/2/rest/dataset>

If no version number is given in the URL then the API defaults to version 1, so this URL will list the site's datasets using API version 1:

<http://demo.ckan.org/api/rest/dataset>

Dataset names can change, so to reliably refer to the same dataset over time, you will want to use the dataset's ID and therefore use API v2. Alternatively, many people prefer to deal with Names, so API v1 suits them.

When posting parameters with your API requests, you can refer to objects by either their name or ID, interchangeably.

Model API

Model resources are available at published locations. They are represented with a variety of data formats. Each resource location supports a number of methods.

The data formats of the requests and the responses are defined below.

Model Resources

Here are the resources of the Model API.

Model Resource	Location
Dataset Register	/rest/dataset
Dataset Entity	/rest/dataset/DATASET-REF
Group Register	/rest/group
Group Entity	/rest/group/GROUP-REF
Tag Register	/rest/tag
Tag Entity	/rest/tag/TAG-NAME
Rating Register	/rest/rating
Dataset Relationships Register	/rest/dataset/DATASET-REF/relationships
Dataset Relationships Register	/rest/dataset/DATASET-REF/RELATIONSHIP-TYPE
Dataset Relationships Register	/rest/dataset/DATASET-REF/relationships/DATASET-REF
Dataset Relationship Entity	/rest/dataset/DATASET-REF/RELATIONSHIP-TYPE/DATASET-REF
Dataset's Revisions Entity	/rest/dataset/DATASET-REF/revisions
Revision Register	/rest/revision
Revision Entity	/rest/revision/REVISION-ID
License List	/rest/licenses

Possible values for DATASET-REF are the dataset id, or the current dataset name.

Possible values for RELATIONSHIP-TYPE are described in the Relationship-Type data format.

Model Methods

Here are the methods of the Model API.

Resource	Method	Request	Response
Dataset Register	GET		Dataset-List
Dataset Register	POST	Dataset	
Dataset Entity	GET		Dataset
Dataset Entity	PUT	Dataset	
Group Register	GET		Group-List
Group Register	POST	Group	
Group Entity	GET		Group
Group Entity	PUT	Group	
Tag Register	GET		Tag-List
Tag Entity	GET		Dataset-List
Rating Register	POST	Rating	
Rating Entity	GET		Rating
Dataset Relationships Register	GET		Pkg-Relationships
Dataset Relationship Entity	GET		Pkg-Relationship
Dataset Relationships Register	POST	Pkg-Relationship	
Dataset Relationship Entity	PUT	Pkg-Relationship	
Dataset's Revisions Entity	GET		Pkg-Revisions
Revision List	GET		Revision-List
Revision Entity	GET		Revision
License List	GET		License-List

In general:

- GET to a register resource will *list* the entities of that type.
- GET of an entity resource will *show* the entity's properties.
- POST of entity data to a register resource will *create* the new entity.

- PUT of entity data to an existing entity resource will *update* it.

It is usually clear whether you are trying to create or update, so in these cases, HTTP POST and PUT methods are accepted by CKAN interchangeably.

Model Formats

Here are the data formats for the Model API:

Name	Format
Dataset-Ref	Dataset-Name-String (API v1) OR Dataset-Id-Uuid (API v2)
Dataset-List	[Dataset-Ref, Dataset-Ref, Dataset-Ref, ...]
Dataset	{ id: Uuid, name: Name-String, title: String, version: String, url: String, resources: [Resource, Resource, ...], author: String, author_email: String, maintainer: String, maintainer_email: String, license_id: String, tags: Tag-List, notes: String, extras: { Name-String: String, ... } } See note below on additional fields upon GET of a dataset.
Group-Ref	Group-Name-String (API v1) OR Group-Id-Uuid (API v2)
Group-List	[Group-Ref, Group-Ref, Group-Ref, ...]
Group	{ name: Group-Name-String, title: String, description: String, packages: Dataset-List }
Tag-List	[Name-String, Name-String, Name-String, ...]
Tag	{ name: Name-String }
Resource	{ url: String, format: String, description: String, hash: String }
Rating	{ dataset: Name-String, rating: int }
Pkg-Relationships	[Pkg-Relationship, Pkg-Relationship, ...]
Pkg-Relationship	{ subject: Dataset-Name-String, object: Dataset-Name-String, type: Relationship-Type, comment: String }
Pkg-Revisions	[Pkg-Revision, Pkg-Revision, Pkg-Revision, ...]
Pkg-Revision	{ id: Uuid, message: String, author: String, timestamp: Date-Time }
Relationship-Type	One of: 'depends_on', 'dependency_of', 'derives_from', 'has_derivation', 'child_of', 'parent_of', 'links_to', 'linked_from'.
Revision-List	[revision_id, revision_id, revision_id, ...]
Revision	{ id: Uuid, message: String, author: String, timestamp: Date-Time, datasets: Dataset-List }
License-List	[License, License, License, ...]
License	{ id: Name-String, title: String, is_okd_compliant: Boolean, is_osi_compliant: Boolean, tags: Tag-List, family: String, url: String, maintainer: String, date_created: Date-Time, status: String }

To send request data, create the JSON-format string (encode in UTF8) put it in the request body and send it using PUT or POST.

Response data will be in the response body in JSON format.

Notes:

- When you update an object, fields that you don't supply will remain as they were before.
- To delete an 'extra' key-value pair, supply the key with JSON value: `null`
- When you read a dataset, some additional information is supplied that you cannot modify and POST back to the CKAN API. These 'read-only' fields are provided only on the Dataset GET. This is a convenience to clients, to

save further requests. This applies to the following fields:

Key	Description
id	Unique Uuid for the Dataset
revision_id	Latest revision ID for the core Package data (but is not affected by changes to tags, groups, extras, relationships etc)
metadata_created	Date the Dataset (record) was created
meta-data_modified	Date the Dataset (record) was last modified
relationships	info on Dataset Relationships
ratings_average	
ratings_count	
ckan_url	full URL of the Dataset
download_url (API v1)	URL of the first Resource
isopen	boolean indication of whether dataset is open according to Open Knowledge Definition, based on other fields
notes_rendered	HTML rendered version of the Notes field (which may contain Markdown)

Search API

Search resources are available at published locations. They are represented with a variety of data formats. Each resource location supports a number of methods.

The data formats of the requests and the responses are defined below.

Search Resources

Here are the published resources of the Search API.

Search Resource	Location
Dataset Search	/search/dataset
Resource Search	/search/resource
Revision Search	/search/revision
Tag Counts	/tag_counts

See below for more information about dataset and revision search parameters.

Search Methods

Here are the methods of the Search API.

Resource	Method	Request	Response
Dataset Search	POST	Dataset-Search-Params	Dataset-Search-Response
Resource Search	POST	Resource-Search-Params	Resource-Search-Response
Revision Search	POST	Revision-Search-Params	Revision-List
Tag Counts	GET		Tag-Count-List

It is also possible to supply the search parameters in the URL of a GET request, for example `/api/search/dataset?q=geodata&allfields=1`.

Search Formats

Here are the data formats for the Search API.

Name	Format
Dataset-Search-Params Resource-Search-Params Revision-Search-Params Dataset-Search-Response Resource-Search-Response Revision-List Tag-Count-List	{ Param-Key: Param-Value, Param-Key: Param-Value, ... } See below for full details of search parameters across the various domain objects. { count: Count-int, results: [Dataset, Dataset, ...] } { count: Count-int, results: [Resource, Resource, ...] } [Revision-Id, Revision-Id, Revision-Id, ...] NB: Ordered with youngest revision first [[Name-String, Integer], [Name-String, Integer], ...]

The `Dataset` and `Revision` data formats are as defined in [Model Formats](#).

Dataset Parameters

Param-Key	Param-Value	Examples	Notes
q	Search-String	q=geodata q=government+sweden q=%22drug%20abuse%22 q=tags:"river pollution"	Criteria to search the dataset fields for. URL-encoded search text. (You can also concatenate words with a '+' symbol in a URL.) Search results must contain all the specified words. You can also search within specific fields. All search parameters can be json-encoded and supplied to this parameter as a more flexible alternative in GET requests. Search in a particular a field.
qjson	JSON encoded options	['q': 'geodata']	
title, tags, notes, groups, author, maintainer, update_frequency, or any 'extra' field name e.g. department	Search-String	title=uk&tags=health department=environment tags=health&tags=pollution tags=river%20pollution	
order_by	field-name (default=rank)	order_by=name	Specify either rank or the field to sort the results by Pagination options. Offset is the number of the first result and limit is the number of results to return. Each matching search result is given as either a dataset name (0) or the full dataset record (1).
offset, limit	result-int (defaults: offset=0, limit=20)	offset=40&limit=20	
all_fields	0 (default) or 1	all_fields=1	

Note: `filter_by_openness` and `filter_by_downloadable` were dropped from CKAN version 1.5 onwards.

Note: Only public datasets can be accessed via the legacy search API, regardless of the provided authorization. If you need to access private datasets via the API you will need to use the `package_search` method of the [API guide](#).

Resource Parameters

Param-Key	Param-Value	Example	Notes
url, format, description	Search-String	url=statistics.org format=xls description=Research+Institute	Criteria to search the dataset fields for. URL-encoded search text. This search string must be found somewhere within the field to match. Case insensitive.
qjson	JSON encoded options	['url': 'www.statistics.org']	All search parameters can be json-encoded and supplied to this parameter as a more flexible alternative in GET requests.
hash	Search-String	hash=b0d7c260-35d4-42ab-9e3d-c1f4db9bc2f0	Searches for an match of the hash field. An exact match or match up to the length of the hash given.
all_fields	0 (default) or 1	all_fields=1	Each matching search result is given as either an ID (0) or the full resource record
offset, limit	result-int (defaults: offset=0, limit=20)	offset=40&limit=20	Pagination options. Offset is the number of the first result and limit is the number of results to return.

Note: Powerful searching from the command-line can be achieved with curl and the qjson parameter. In this case you need to remember to escapt the curly braces and use url encoding (e.g. spaces become %20). For example:

```
curl 'http://thedatahub.org/api/search/dataset?qjson=\{"author": "The%20Stationery%20Office%20Limited"
```

Revision Parameters

Param-Key	Param-Value	Example	Notes
since_time	Date-Time	since_time=2010-05-05T19:42:45.854533	The time can be less precisely stated (e.g 2010-05-05).
since_id	Uuid	since_id=6c9f32ef-1f93-4b2f-891b-fd01924ebe08	The stated id will not be included in the results.

Util API

The Util API provides various utility APIs – e.g. auto-completion APIs used by front-end javascript.

All Util APIs are read-only. The response format is JSON. Javascript calls may want to use the JSONP formatting.

dataset autocomplete

There an autocomplete API for package names which matches on name or title.

This URL:

```
/api/2/util/dataset/autocomplete?incomplete=a%20novel
```

Returns:

```
{"ResultSet": {"Result": [{"match_field": "title", "match_displayed": "A Novel By Tolstoy (annakaren
```

tag autocomplete

There is also an autocomplete API for tags which looks like this:

This URL:

```
/api/2/util/tag/autocomplete?incomplete=ru
```

Returns:

```
{"ResultSet": {"Result": [{"Name": "russian"}]}}
```

resource format autocomplete

Similarly, there is an autocomplete API for the resource format field which is available at:

```
/api/2/util/resource/format_autocomplete?incomplete=cs
```

This returns:

```
{"ResultSet": {"Result": [{"Format": "csv"}]}}
```

markdown

Takes a raw markdown string and returns a corresponding chunk of HTML. CKAN uses the basic Markdown format with some modifications (for security) and useful additions (e.g. auto links to datasets etc. e.g. `dataset:river-quality`).

Example:

```
/api/util/markdown?q=<http://ibm.com/>
```

Returns:

```
"<p><a href="http://ibm.com/" target="_blank" rel="nofollow">http://ibm.com/</a>\n</p>"
```

is slug valid

Checks a name is valid for a new dataset (package) or group, with respect to it being used already.

Example:

```
/api/2/util/is_slug_valid?slug=river-quality&type=package
```

Response:

```
{"valid": true}
```

munge package name

For taking an readable identifier and munging it to ensure it is a valid dataset id. Symbols and whitespace are converted into dashes. Example:

```
/api/util/dataset/munge_name?name=police%20spending%20figures%202009
```

Returns:

```
"police-spending-figures-2009"
```

munge title to package name

For taking a title of a package and munging it to a readable and valid dataset id. Symbols and whitespace are converted into dashes, with multiple dashes collapsed. Ensures that long titles with a year at the end preserves the year should it need to be shortened. Example:

```
/api/util/dataset/munge_title_to_name?title=police:%20spending%20figures%202009
```

Returns:

```
"police-spending-figures-2009"
```

munge tag

For taking a readable word/phrase and munging it to a valid tag (name). Symbols and whitespace are converted into dashes. Example:

```
/api/util/tag/munge?tag=water%20quality
```

Returns:

```
"water-quality"
```

Status Codes

Standard HTTP status codes are used to signal method outcomes.

Code	Name
200	OK
201	OK and new object created (referred to in the Location header)
301	Moved Permanently
400	Bad Request
403	Not Authorized
404	Not Found
409	Conflict (e.g. name already exists)
500	Service Error

Making an API request

To call the CKAN API, post a JSON dictionary in an HTTP POST request to one of CKAN's API URLs. The parameters for the API function should be given in the JSON dictionary. CKAN will also return its response in a JSON dictionary.

One way to post a JSON dictionary to a URL is using the command-line HTTP client [HTTPIe](#). For example, to get a list of the names of all the datasets in the `data-explorer` group on `demo.ckan.org`, install `HTTPIe` and then call the `group_list` API function by running this command in a terminal:

```
http http://demo.ckan.org/api/3/action/group_list id=data-explorer
```

The response from CKAN will look like this:

```
{
  "help": "...",
  "result": [
    "data-explorer",
    "department-of-ricky",
    "geo-examples",
    "geothermal-data",
    "reykjavik",
    "skeenawild-conservation-trust"
  ],
  "success": true
}
```

The response is a JSON dictionary with three keys:

1. `"success": true or false.`

The API aims to always return 200 OK as the status code of its HTTP response, whether there were errors with the request or not, so it's important to always check the value of the `"success"` key in the response dictionary and (if success is false) check the value of the `"error"` key.

Note: If there are major formatting problems with a request to the API, CKAN may still return an HTTP response with a 409, 400 or 500 status code (in increasing order of severity). In future CKAN versions we intend to remove these responses, and instead send a 200 OK response and use the `"success"` and `"error"` items.

2. `"result":` the returned result from the function you called. The type and value of the result depend on which function you called. In the case of the `group_list` function it's a list of strings, the names of all the datasets that belong to the group.

If there was an error responding to your request, the dictionary will contain an `"error"` key with details of the error instead of the `"result"` key. A response dictionary containing an error will look like this:

```
{
  "help": "Creates a package",
  "success": false,
  "error": {
    "message": "Access denied",
    "__type": "Authorization Error"
  }
}
```

3. `"help":` the documentation string for the function you called.

The same HTTP request can be made using Python's standard `urllib2` module, with this Python code:

```
#!/usr/bin/env python
import urllib2
import urllib
import json
import pprint

# Use the json module to dump a dictionary to a string for posting.
```

```

data_string = urllib.quote(json.dumps({'id': 'data-explorer'}))

# Make the HTTP request.
response = urllib2.urlopen('http://demo.ckan.org/api/3/action/group_list',
                           data_string)
assert response.code == 200

# Use the json module to load CKAN's response into a dictionary.
response_dict = json.loads(response.read())

# Check the contents of the response.
assert response_dict['success'] is True
result = response_dict['result']
pprint.pprint(result)

```

Example: Importing datasets with the CKAN API

You can add datasets using CKAN's web interface, but when importing many datasets it's usually more efficient to automate the process in some way. In this example, we'll show you how to use the CKAN API to write a Python script to import datasets into CKAN.

Todo

Make this script more interesting (eg. read data from a CSV file), and all put the script in a .py file somewhere with tests and import it here.

```

#!/usr/bin/env python
import urllib2
import urllib
import json
import pprint

# Put the details of the dataset we're going to create into a dict.
dataset_dict = {
    'name': 'my_dataset_name',
    'notes': 'A long description of my dataset',
}

# Use the json module to dump the dictionary to a string for posting.
data_string = urllib.quote(json.dumps(dataset_dict))

# We'll use the package_create function to create a new dataset.
request = urllib2.Request(
    'http://www.my_ckan_site.com/api/action/package_create')

# Creating a dataset requires an authorization header.
# Replace *** with your API key, from your user account on the CKAN site
# that you're creating the dataset on.
request.add_header('Authorization', '***')

# Make the HTTP request.
response = urllib2.urlopen(request, data_string)
assert response.code == 200

# Use the json module to load CKAN's response into a dictionary.

```

```
response_dict = json.loads(response.read())
assert response_dict['success'] is True

# package_create returns the created package as its result.
created_package = response_dict['result']
pprint.pprint(created_package)
```

For more examples, see *API Examples*.

API versions

The CKAN APIs are versioned. If you make a request to an API URL without a version number, CKAN will choose the latest version of the API:

```
http://demo.ckan.org/api/action/package_list
```

Alternatively, you can specify the desired API version number in the URL that you request:

```
http://demo.ckan.org/api/3/action/package_list
```

Version 3 is currently the only version of the Action API.

We recommend that you specify the API number in your requests, because this ensures that your API client will work across different sites running different version of CKAN (and will keep working on the same sites, when those sites upgrade to new versions of CKAN). Because the latest version of the API may change when a site is upgraded to a new version of CKAN, or may differ on different sites running different versions of CKAN, the result of an API request that doesn't specify the API version number cannot be relied on.

Authentication and API keys

Some API functions require authorization. The API uses the same authorization functions and configuration as the web interface, so if a user is authorized to do something in the web interface they'll be authorized to do it via the API as well.

When calling an API function that requires authorization, you must authenticate yourself by providing your API key with your HTTP request. To find your API key, login to the CKAN site using its web interface and visit your user profile page.

To provide your API key in an HTTP request, include it in either an `Authorization` or `X-CKAN-API-Key` header. (The name of the HTTP header can be configured with the `apikey_header_name` option in your CKAN configuration file.)

For example, to ask whether or not you're currently following the user `markw` on `demo.ckan.org` using `HTTPIe`, run this command:

```
http http://demo.ckan.org/api/3/action/am_following_user id=markw Authorization:XXX
```

(Replacing `XXX` with your API key.)

Or, to get the list of activities from your user dashboard on `demo.ckan.org`, run this Python code:

```
request = urllib2.Request('http://demo.ckan.org/api/3/action/dashboard_activity_list')
request.add_header('Authorization', 'XXX')
response_dict = json.loads(urllib2.urlopen(request, '{}').read())
```

GET-able API functions

Functions defined in `ckan.logic.action.get` can also be called with an HTTP GET request. For example, to get the list of datasets (packages) from `demo.ckan.org`, open this URL in your browser:

`http://demo.ckan.org/api/3/action/package_list`

Or, to search for datasets (packages) matching the search query `spending`, on `demo.ckan.org`, open this URL in your browser:

`http://demo.ckan.org/api/3/action/package_search?q=spending`

Tip: Browser plugins like [JSONView for Firefox](#) or [Chrome](#) will format and color CKAN's JSON response nicely in your browser.

The search query is given as a URL parameter `?q=spending`. Multiple URL parameters can be appended, separated by `&` characters, for example to get only the first 10 matching datasets open this URL:

`http://demo.ckan.org/api/3/action/package_search?q=spending&rows=10`

When an action requires a list of strings as the value of a parameter, the value can be sent by giving the parameter multiple times in the URL:

`http://demo.ckan.org/api/3/action/term_translation_show?terms=russian&terms=romantic%20novel`

JSONP support

To cater for scripts from other sites that wish to access the API, the data can be returned in JSONP format, where the JSON data is 'padded' with a function call. The function is named in the 'callback' parameter. For example:

`http://demo.ckan.org/api/3/action/package_show?id=adur_district_spending&callback=myfunction`

Todo

This doesn't work with all functions.

API Examples

Tags (not in a vocabulary)

A list of all tags:

- browser: `http://demo.ckan.org/api/3/action/tag_list`
- curl: `curl http://demo.ckan.org/api/3/action/tag_list`
- ckanapi: `ckanapi -r http://demo.ckan.org action tag_list`

Top 10 tags used by datasets:

- browser: `http://demo.ckan.org/api/action/package_search?facet.field=[%22tags%22]&facet.limit=10&rows=0`
- curl: `curl 'http://demo.ckan.org/api/action/package_search?facet.field=["tags"]&facet.limit=10&rows=0'`
- ckanapi: `ckanapi -r http://demo.ckan.org action package_search facet.field='["tags"]' facet.limit=10 rows=0`

All datasets that have tag 'economy':

- browser: http://demo.ckan.org/api/3/action/package_search?fq=tags:economy
- curl: `curl 'http://demo.ckan.org/api/3/action/package_search?fq=tags:economy'`
- ckanapi: `ckanapi -r http://demo.ckan.org action package_search fq='tags:economy'`

Tag Vocabularies

Top 10 tags and vocabulary tags used by datasets:

- browser: [http://demo.ckan.org/api/action/package_search?facet.field=\[%22tags%22\]&facet.limit=10&rows=0](http://demo.ckan.org/api/action/package_search?facet.field=[%22tags%22]&facet.limit=10&rows=0)
- curl: `curl 'http://demo.ckan.org/api/action/package_search?facet.field=["tags"]&facet.limit=10&rows=0'`
- ckanapi: `ckanapi -r http://demo.ckan.org action package_search facet.field='["tags"]' facet.limit=10 rows=0`

e.g. Facet: *vocab_Topics* means there is a vocabulary called Topics, and its top tags are listed under it.

A list of datasets using tag 'education' from vocabulary 'Topics':

- browser: https://data.hdx.rwlab.org/api/3/action/package_search?fq=vocab_Topics:education
- curl: `curl 'https://data.hdx.rwlab.org/api/3/action/package_search?fq=vocab_Topics:education'`
- ckanapi: `ckanapi -r https://data.hdx.rwlab.org action package_search fq='vocab_Topics:education'`

Uploading a new version of a resource file

You can use the `upload` parameter of the `resource_update()` function to upload a new version of a resource file. This requires a `multipart/form-data` request, with `httpie` you can do this using the `@file.csv`:

```
http --json POST http://demo.ckan.org/api/3/action/resource_update id=<resource id> upload=@updated_file.csv
```

Action API reference

Note: If you call one of the action functions listed below and the function raises an exception, the API will return a JSON dictionary with keys `"success": false` and an `"error"` key indicating the exception that was raised.

For example `member_list()` (which returns a list of the members of a group) raises `NotFound` if the group doesn't exist. If you called it over the API, you'd get back a JSON dict like this:

```
{
  "success": false
  "error": {
    "__type": "Not Found Error",
    "message": "Not found"
  },
  "help": "...",
}
```

ckan.logic.action.get

API functions for searching for and getting data from CKAN.

`ckan.logic.action.get.site_read(context, data_dict=None)`
Return True.

Return type boolean

`ckan.logic.action.get.package_list(context, data_dict)`
Return a list of the names of the site's datasets (packages).

Parameters

- **limit** (*int*) – if given, the list of datasets will be broken into pages of at most `limit` datasets per page and only one page will be returned at a time (optional)
- **offset** (*int*) – when `limit` is given, the offset to start returning packages from

Return type list of strings

`ckan.logic.action.get.current_package_list_with_resources(context, data_dict)`
Return a list of the site's datasets (packages) and their resources.

The list is sorted most-recently-modified first.

Parameters

- **limit** (*int*) – if given, the list of datasets will be broken into pages of at most `limit` datasets per page and only one page will be returned at a time (optional)
- **offset** (*int*) – when `limit` is given, the offset to start returning packages from
- **page** (*int*) – when `limit` is given, which page to return, Deprecated: use `offset`

Return type list of dictionaries

`ckan.logic.action.get.revision_list(context, data_dict)`
Return a list of the IDs of the site's revisions.

Return type list of strings

`ckan.logic.action.get.package_revision_list(context, data_dict)`
Return a dataset (package)'s revisions as a list of dictionaries.

Parameters **id** (*string*) – the id or name of the dataset

`ckan.logic.action.get.related_show(context, data_dict=None)`
Return a single related item.

Parameters **id** (*string*) – the id of the related item to show

Return type dictionary

`ckan.logic.action.get.related_list(context, data_dict=None)`
Return a dataset's related items.

Parameters

- **id** (*string*) – id or name of the dataset (optional)
- **dataset** (*dictionary*) – dataset dictionary of the dataset (optional)
- **type_filter** (*string*) – the type of related item to show (optional, default: None, show all items)

- **sort** (*string*) – the order to sort the related items in, possible values are ‘view_count_asc’, ‘view_count_desc’, ‘created_asc’ or ‘created_desc’ (optional)
- **featured** (*bool*) – whether or not to restrict the results to only featured related items (optional, default: False)

Return type list of dictionaries

`ckan.logic.action.get.member_list(context, data_dict=None)`

Return the members of a group.

The user must have permission to ‘get’ the group.

Parameters

- **id** (*string*) – the id or name of the group
- **object_type** (*string*) – restrict the members returned to those of a given type, e.g. ‘user’ or ‘package’ (optional, default: None)
- **capacity** (*string*) – restrict the members returned to those with a given capacity, e.g. ‘member’, ‘editor’, ‘admin’, ‘public’, ‘private’ (optional, default: None)

Return type list of (id, type, capacity) tuples

Raises `ckan.logic.NotFound`: if the group doesn’t exist

`ckan.logic.action.get.group_list(context, data_dict)`

Return a list of the names of the site’s groups.

Parameters

- **order_by** (*string*) – the field to sort the list by, must be ‘name’ or ‘packages’ (optional, default: ‘name’) Deprecated use sort.
- **sort** (*string*) – sorting of the search results. Optional. Default: “name asc” string of field name and sort-order. The allowed fields are ‘name’, ‘package_count’ and ‘title’
- **limit** (*int*) – if given, the list of groups will be broken into pages of at most `limit` groups per page and only one page will be returned at a time (optional)
- **offset** (*int*) – when `limit` is given, the offset to start returning groups from
- **groups** (*list of strings*) – a list of names of the groups to return, if given only groups whose names are in this list will be returned (optional)
- **all_fields** (*boolean*) – return group dictionaries instead of just names. Only core fields are returned - get some more using the `include_*` options. Returning a list of packages is too expensive, so the `packages` property for each group is deprecated, but there is a count of the packages in the `package_count` property. (optional, default: False)
- **include_extras** (*boolean*) – if `all_fields`, include the group extra fields (optional, default: False)
- **include_tags** (*boolean*) – if `all_fields`, include the group tags (optional, default: False)
- **include_groups** (*boolean*) – if `all_fields`, include the groups the groups are in (optional, default: False).
- **include_users** (*boolean*) – if `all_fields`, include the group users (optional, default: False).

Return type list of strings

`ckan.logic.action.get.organization_list(context, data_dict)`

Return a list of the names of the site’s organizations.

Parameters

- **order_by** (*string*) – the field to sort the list by, must be 'name' or 'packages' (optional, default: 'name') Deprecated use sort.
- **sort** (*string*) – sorting of the search results. Optional. Default: "name asc" string of field name and sort-order. The allowed fields are 'name', 'package_count' and 'title'
- **limit** (*int*) – if given, the list of organizations will be broken into pages of at most `limit` organizations per page and only one page will be returned at a time (optional)
- **offset** (*int*) – when `limit` is given, the offset to start returning organizations from
- **organizations** (*list of strings*) – a list of names of the groups to return, if given only groups whose names are in this list will be returned (optional)
- **all_fields** (*boolean*) – return group dictionaries instead of just names. Only core fields are returned - get some more using the `include_*` options. Returning a list of packages is too expensive, so the `packages` property for each group is deprecated, but there is a count of the packages in the `package_count` property. (optional, default: `False`)
- **include_extras** (*boolean*) – if `all_fields`, include the organization extra fields (optional, default: `False`)
- **include_tags** (*boolean*) – if `all_fields`, include the organization tags (optional, default: `False`)
- **include_groups** – if `all_fields`, include the organizations the organizations are in (optional, default: `False`)
- **include_users** (*boolean*) – if `all_fields`, include the organization users (optional, default: `False`).

Return type list of strings`ckan.logic.action.get.group_list_authz(context, data_dict)`

Return the list of groups that the user is authorized to edit.

Parameters

- **available_only** (*boolean*) – remove the existing groups in the package (optional, default: `False`)
- **am_member** – if `True` return only the groups the logged-in user is a member of, otherwise return all groups that the user is authorized to edit (for example, sysadmin users are authorized to edit all groups) (optional, default: `False`)

Returns list of dictized groups that the user is authorized to edit**Return type** list of dicts`ckan.logic.action.get.organization_list_for_user(context, data_dict)`

Return the organizations that the user has a given permission for.

By default this returns the list of organizations that the currently authorized user can edit, i.e. the list of organizations that the user is an admin of.

Specifically it returns the list of organizations that the currently authorized user has a given permission (for example: "manage_group") against.

When a user becomes a member of an organization in CKAN they're given a "capacity" (sometimes called a "role"), for example "member", "editor" or "admin".

Each of these roles has certain permissions associated with it. For example the admin role has the “admin” permission (which means they have permission to do anything). The editor role has permissions like “create_dataset”, “update_dataset” and “delete_dataset”. The member role has the “read” permission.

This function returns the list of organizations that the authorized user has a given permission for. For example the list of organizations that the user is an admin of, or the list of organizations that the user can create datasets in. This takes account of when permissions cascade down an organization hierarchy.

Parameters **permission** (*string*) – the permission the user has against the returned organizations, for example “read” or “create_dataset” (optional, default: “edit_group”)

Returns list of organizations that the user has the given permission for

Return type list of dicts

`ckan.logic.action.get.group_revision_list(context, data_dict)`

Return a group’s revisions.

Parameters **id** (*string*) – the name or id of the group

Return type list of dictionaries

`ckan.logic.action.get.organization_revision_list(context, data_dict)`

Return an organization’s revisions.

Parameters **id** (*string*) – the name or id of the organization

Return type list of dictionaries

`ckan.logic.action.get.license_list(context, data_dict)`

Return the list of licenses available for datasets on the site.

Return type list of dictionaries

`ckan.logic.action.get.tag_list(context, data_dict)`

Return a list of the site’s tags.

By default only free tags (tags that don’t belong to a vocabulary) are returned. If the `vocabulary_id` argument is given then only tags belonging to that vocabulary will be returned instead.

Parameters

- **query** (*string*) – a tag name query to search for, if given only tags whose names contain this string will be returned (optional)
- **vocabulary_id** (*string*) – the id or name of a vocabulary, if give only tags that belong to this vocabulary will be returned (optional)
- **all_fields** (*boolean*) – return full tag dictionaries instead of just names (optional, default: `False`)

Return type list of dictionaries

`ckan.logic.action.get.user_list(context, data_dict)`

Return a list of the site’s user accounts.

Parameters

- **q** (*string*) – restrict the users returned to those whose names contain a string (optional)
- **order_by** (*string*) – which field to sort the list by (optional, default: ‘name’). Can be any user field or `edits` (i.e. `number_of_edits`).

Return type list of user dictionaries. User properties include: `number_of_edits` which counts the revisions by the user and `number_created_packages` which excludes datasets which are private or draft state.

`ckan.logic.action.get.package_relationships_list(context, data_dict)`

Return a dataset (package)'s relationships.

Parameters

- **id** (*string*) – the id or name of the first package
- **id2** – the id or name of the second package
- **rel** – relationship as string see `package_relationship_create()` for the relationship types (optional)

Return type list of dictionaries

`ckan.logic.action.get.package_show(context, data_dict)`

Return the metadata of a dataset (package) and its resources.

Parameters

- **id** (*string*) – the id or name of the dataset
- **use_default_schema** (*bool*) – use default package schema instead of a custom schema defined with an IDatasetForm plugin (default: False)
- **include_tracking** (*bool*) – add tracking information to dataset and resources (default: False)

Return type dictionary

`ckan.logic.action.get.resource_show(context, data_dict)`

Return the metadata of a resource.

Parameters

- **id** (*string*) – the id of the resource
- **include_tracking** (*bool*) – add tracking information to dataset and resources (default: False)

Return type dictionary

`ckan.logic.action.get.resource_view_show(context, data_dict)`

Return the metadata of a resource_view.

Parameters **id** (*string*) – the id of the resource_view

Return type dictionary

`ckan.logic.action.get.resource_view_list(context, data_dict)`

Return the list of resource views for a particular resource.

Parameters **id** (*string*) – the id of the resource

Return type list of dictionaries.

`ckan.logic.action.get.resource_status_show(context, data_dict)`

Return the statuses of a resource's tasks.

Parameters **id** (*string*) – the id of the resource

Return type list of (status, date_done, traceback, task_status) dictionaries

`ckan.logic.action.get.revision_show(context, data_dict)`

Return the details of a revision.

Parameters **id** (*string*) – the id of the revision

Return type dictionary

`ckan.logic.action.get.group_show(context, data_dict)`

Return the details of a group.

Parameters

- **id** (*boolean*) – the id or name of the group
- **include_datasets** – include a list of the group’s datasets (optional, default: `False`)
- **include_extras** – include the group’s extra fields (optional, default: `True`)
- **include_users** – include the group’s users (optional, default: `True`)
- **include_groups** – include the group’s sub groups (optional, default: `True`)
- **include_tags** – include the group’s tags (optional, default: `True`)
- **include_followers** – include the group’s number of followers (optional, default: `True`)

Return type dictionary

Note: Only its first 1000 datasets are returned

`ckan.logic.action.get.organization_show(context, data_dict)`

Return the details of an organization.

Parameters

- **id** (*boolean*) – the id or name of the organization
- **include_datasets** – include a list of the organization’s datasets (optional, default: `False`)
- **include_extras** – include the organization’s extra fields (optional, default: `True`)
- **include_users** – include the organization’s users (optional, default: `True`)
- **include_groups** – include the organization’s sub groups (optional, default: `True`)
- **include_tags** – include the organization’s tags (optional, default: `True`)
- **include_followers** – include the organization’s number of followers (optional, default: `True`)

Return type dictionary

Note: Only its first 1000 datasets are returned

`ckan.logic.action.get.group_package_show(context, data_dict)`

Return the datasets (packages) of a group.

Parameters

- **id** (*string*) – the id or name of the group
- **limit** (*int*) – the maximum number of datasets to return (optional)

Return type list of dictionaries

`ckan.logic.action.get.tag_show(context, data_dict)`

Return the details of a tag and all its datasets.

Parameters

- **id** (*string*) – the name or id of the tag
- **vocabulary_id** (*string*) – the id or name of the tag vocabulary that the tag is in - if it is not specified it will assume it is a free tag. (optional)

- **include_datasets** (*bool*) – include a list of the tag’s datasets. (Up to a limit of 1000 - for more flexibility, use `package_search` - see `package_search()` for an example.) (optional, default: `False`)

Returns the details of the tag, including a list of all of the tag’s datasets and their details

Return type dictionary

`ckan.logic.action.get.user_show(context, data_dict)`

Return a user account.

Either the `id` or the `user_obj` parameter must be given.

Parameters

- **id** (*string*) – the id or name of the user (optional)
- **user_obj** (*user dictionary*) – the user dictionary of the user (optional)
- **include_datasets** (*boolean*) – Include a list of datasets the user has created. If it is the same user or a sysadmin requesting, it includes datasets that are draft or private. (optional, default:`False`, limit:50)
- **include_num_followers** (*boolean*) – Include the number of followers the user has (optional, default:`False`)

Returns the details of the user. Includes `email_hash`, `number_of_edits` and `number_created_packages` (which excludes draft or private datasets unless it is the same user or sysadmin making the request). Excludes the password (hash) and `reset_key`. If it is the same user or a sysadmin requesting, the email and apikey are included.

Return type dictionary

`ckan.logic.action.get.package_autocomplete(context, data_dict)`

Return a list of datasets (packages) that match a string.

Datasets with names or titles that contain the query string will be returned.

Parameters

- **q** (*string*) – the string to search for
- **limit** (*int*) – the maximum number of resource formats to return (optional, default: 10)

Return type list of dictionaries

`ckan.logic.action.get.format_autocomplete(context, data_dict)`

Return a list of resource formats whose names contain a string.

Parameters

- **q** (*string*) – the string to search for
- **limit** (*int*) – the maximum number of resource formats to return (optional, default: 5)

Return type list of strings

`ckan.logic.action.get.user_autocomplete(context, data_dict)`

Return a list of user names that contain a string.

Parameters

- **q** (*string*) – the string to search for
- **limit** (*int*) – the maximum number of user names to return (optional, default: 20)

Return type a list of user dictionaries each with keys `'name'`, `'fullname'`, and `'id'`

`ckan.logic.action.get.organization_autocomplete(context, data_dict)`

Return a list of organization names that contain a string.

Parameters

- **q** (*string*) – the string to search for
- **limit** (*int*) – the maximum number of organizations to return (optional, default: 20)

Return type a list of organization dictionaries each with keys 'name', 'title', and 'id'

`ckan.logic.action.get.package_search(context, data_dict)`

Searches for packages satisfying a given search criteria.

This action accepts solr search query parameters (details below), and returns a dictionary of results, including dictized datasets that match the search criteria, a search count and also facet information.

Solr Parameters:

For more in depth treatment of each paramter, please read the [Solr Documentation](#).

This action accepts a *subset* of solr's search query parameters:

Parameters

- **q** (*string*) – the solr query. Optional. Default: "*" : *
- **fq** (*string*) – any filter queries to apply. Note: `+site_id:{ckan_site_id}` is added to this string prior to the query being executed.
- **sort** (*string*) – sorting of the search results. Optional. Default: 'relevance asc, metadata_modified desc'. As per the solr documentation, this is a comma-separated string of field names and sort-orderings.
- **rows** (*int*) – the number of matching rows to return.
- **start** (*int*) – the offset in the complete result for where the set of returned datasets should begin.
- **facet** (*string*) – whether to enable faceted results. Default: True.
- **facet.mincount** (*int*) – the minimum counts for facet fields should be included in the results.
- **facet.limit** (*int*) – the maximum number of values the facet fields return. A negative value means unlimited. This can be set instance-wide with the `search.facets.limit` config option. Default is 50.
- **facet.field** (*list of strings*) – the fields to facet upon. Default empty. If empty, then the returned facet information is empty.
- **include_drafts** (*boolean*) – if True, draft datasets will be included in the results. A user will only be returned their own draft datasets, and a sysadmin will be returned all draft datasets. Optional, the default is False.
- **use_default_schema** (*bool*) – use default package schema instead of a custom schema defined with an IDatasetForm plugin (default: False)

The following advanced Solr parameters are supported as well. Note that some of these are only available on particular Solr versions. See Solr's [dismax](#) and [edismax](#) documentation for further details on them:

`qf, wt, bf, boost, tie, defType, mm`

Examples:

`q=flood` datasets containing the word *flood*, *floods* or *flooding* `fq=tags:economy` datasets with the tag *economy* `facet.field=["tags"]` `facet.limit=10` `rows=0` top 10 tags

Results:

The result of this action is a dict with the following keys:

Return type A dictionary with the following keys

Parameters

- **count** (*int*) – the number of results found. Note, this is the total number of results found, not the total number of results returned (which is affected by limit and row parameters used in the input).
- **results** (*list of dictized datasets.*) – ordered list of datasets matching the query, where the ordering defined by the sort parameter used in the query.
- **facets** (*DEPRECATED dict*) – DEPRECATED. Aggregated information about facet counts.
- **search_facets** (*nested dict of dicts.*) – aggregated information about facet counts. The outer dict is keyed by the facet field name (as used in the search query). Each entry of the outer dict is itself a dict, with a “title” key, and an “items” key. The “items” key’s value is a list of dicts, each with “count”, “display_name” and “name” entries. The display_name is a form of the name that can be used in titles.

An example result:

```
{'count': 2,
 'results': [ { <snip> }, { <snip> }],
 'search_facets': {'u'tags': {'items': [{'count': 1,
                                         'display_name': u'tolstoy',
                                         'name': u'tolstoy'},
                                       {'count': 2,
                                         'display_name': u'russian',
                                         'name': u'russian'}
                                     ]}
                 }
```

Limitations:

The full solr query language is not exposed, including.

- ¶ The parameter that controls which fields are returned in the solr query cannot be changed. CKAN always returns the matched datasets as dictionary objects.

`ckan.logic.action.get.resource_search(context, data_dict)`

Searches for resources satisfying a given search criteria.

It returns a dictionary with 2 fields: `count` and `results`. The `count` field contains the total number of Resources found without the limit or query parameters having an effect. The `results` field is a list of dictized Resource objects.

The ‘query’ parameter is a required field. It is a string of the form `{field}:{term}` or a list of strings, each of the same form. Within each string, `{field}` is a field or extra field on the Resource domain object.

If `{field}` is “hash”, then an attempt is made to match the `{term}` as a *prefix* of the `Resource.hash` field.

If `{field}` is an extra field, then an attempt is made to match against the extra fields stored against the Resource.

Note: The search is limited to search against extra fields declared in the config setting `ckan.extra_resource_fields`.

Note: Due to a Resource's extra fields being stored as a json blob, the match is made against the json string representation. As such, false positives may occur:

If the search criteria is:

```
query = "field1:term1"
```

Then a json blob with the string representation of:

```
{"field1": "foo", "field2": "term1"}
```

will match the search criteria! This is a known short-coming of this approach.

All matches are made ignoring case; and apart from the "hash" field, a term matches if it is a substring of the field's value.

Finally, when specifying more than one search criteria, the criteria are AND-ed together.

The `order` parameter is used to control the ordering of the results. Currently only ordering one field is available, and in ascending order only.

The `fields` parameter is deprecated as it is not compatible with calling this action with a GET request to the action API.

The context may contain a flag, `search_query`, which if True will make this action behave as if being used by the internal search api. ie - the results will not be dictized, and SearchErrors are thrown for bad search queries (rather than ValidationErrors).

Parameters

- **query** (string or list of strings of the form `{field}:{term1}`) – The search criteria. See above for description.
- **fields** (*dict of fields to search terms.*) – Deprecated
- **order_by** (*string*) – A field on the Resource model that orders the results.
- **offset** (*int*) – Apply an offset to the query.
- **limit** (*int*) – Apply a limit to the query.

Returns A dictionary with a `count` field, and a `results` field.

Return type dict

`ckan.logic.action.get.tag_search(context, data_dict)`

Return a list of tags whose names contain a given string.

By default only free tags (tags that don't belong to any vocabulary) are searched. If the `vocabulary_id` argument is given then only tags belonging to that vocabulary will be searched instead.

Parameters

- **query** (*string or list of strings*) – the string(s) to search for
- **vocabulary_id** (*string*) – the id or name of the tag vocabulary to search in (optional)
- **fields** (*dictionary*) – deprecated
- **limit** (*int*) – the maximum number of tags to return
- **offset** (*int*) – when `limit` is given, the offset to start returning tags from

Returns

A dictionary with the following keys:

'**count**' The number of tags in the result.

'results' The list of tags whose names contain the given string, a list of dictionaries.

Return type dictionary

`ckan.logic.action.get.tag_autocomplete(context, data_dict)`

Return a list of tag names that contain a given string.

By default only free tags (tags that don't belong to any vocabulary) are searched. If the `vocabulary_id` argument is given then only tags belonging to that vocabulary will be searched instead.

Parameters

- **query** (*string*) – the string to search for
- **vocabulary_id** (*string*) – the id or name of the tag vocabulary to search in (optional)
- **fields** (*dictionary*) – deprecated
- **limit** (*int*) – the maximum number of tags to return
- **offset** (*int*) – when `limit` is given, the offset to start returning tags from

Return type list of strings

`ckan.logic.action.get.task_status_show(context, data_dict)`

Return a task status.

Either the `id` parameter *or* the `entity_id`, `task_type` *and* `key` parameters must be given.

Parameters

- **id** (*string*) – the id of the task status (optional)
- **entity_id** (*string*) – the `entity_id` of the task status (optional)
- **task_type** – the `task_type` of the task status (optional)
- **key** (*string*) – the key of the task status (optional)

Return type dictionary

`ckan.logic.action.get.term_translation_show(context, data_dict)`

Return the translations for the given term(s) and language(s).

Parameters

- **terms** (*list of strings*) – the terms to search for translations of, e.g. `'Russian'`, `'romantic novel'`
- **lang_codes** (*list of language code strings*) – the language codes of the languages to search for translations into, e.g. `'en'`, `'de'` (optional, default is to search for translations into any language)

Return type a list of term translation dictionaries each with keys `'term'` (the term searched for, in the source language), `'term_translation'` (the translation of the term into the target language) and `'lang_code'` (the language code of the target language)

`ckan.logic.action.get.get_site_user(context, data_dict)`

Return the ckan site user

Parameters **defer_commit** (*boolean*) – by default (or if set to false) `get_site_user` will commit and clean up the current transaction. If set to true, caller is responsible for committing transaction after `get_site_user` is called. Leaving open connections can cause cli commands to hang! (optional, default: False)

`ckan.logic.action.get.roles_show(context, data_dict)`

Return the roles of all users and authorization groups for an object.

Parameters

- **domain_object** (*string*) – a package or group name or id to filter the results by
- **user** (*string*) – a user name or id

Return type list of dictionaries

`ckan.logic.action.get.status_show(context, data_dict)`

Return a dictionary with information about the site's configuration.

Return type dictionary

`ckan.logic.action.get.vocabulary_list(context, data_dict)`

Return a list of all the site's tag vocabularies.

Return type list of dictionaries

`ckan.logic.action.get.vocabulary_show(context, data_dict)`

Return a single tag vocabulary.

Parameters **id** (*string*) – the id or name of the vocabulary

Returns the vocabulary.

Return type dictionary

`ckan.logic.action.get.user_activity_list(context, data_dict)`

Return a user's public activity stream.

You must be authorized to view the user's profile.

Parameters

- **id** (*string*) – the id or name of the user
- **offset** (*int*) – where to start getting activity items from (optional, default: 0)
- **limit** (*int*) – the maximum number of activities to return (optional, default: 31, the default value is configurable via the `ckan.activity_list_limit` setting)

Return type list of dictionaries

`ckan.logic.action.get.package_activity_list(context, data_dict)`

Return a package's activity stream.

You must be authorized to view the package.

Parameters

- **id** (*string*) – the id or name of the package
- **offset** (*int*) – where to start getting activity items from (optional, default: 0)
- **limit** (*int*) – the maximum number of activities to return (optional, default: 31, the default value is configurable via the `ckan.activity_list_limit` setting)

Return type list of dictionaries

`ckan.logic.action.get.group_activity_list(context, data_dict)`

Return a group's activity stream.

You must be authorized to view the group.

Parameters

- **id** (*string*) – the id or name of the group
- **offset** (*int*) – where to start getting activity items from (optional, default: 0)

- **limit** (*int*) – the maximum number of activities to return (optional, default: 31, the default value is configurable via the `ckan.activity_list_limit` setting)

Return type list of dictionaries

`ckan.logic.action.get.organization_activity_list` (*context*, *data_dict*)

Return a organization's activity stream.

Parameters **id** (*string*) – the id or name of the organization

Return type list of dictionaries

`ckan.logic.action.get.recently_changed_packages_activity_list` (*context*,
data_dict)

Return the activity stream of all recently added or changed packages.

Parameters

- **offset** (*int*) – where to start getting activity items from (optional, default: 0)
- **limit** (*int*) – the maximum number of activities to return (optional, default: 31, the default value is configurable via the `ckan.activity_list_limit` setting)

Return type list of dictionaries

`ckan.logic.action.get.activity_detail_list` (*context*, *data_dict*)

Return an activity's list of activity detail items.

Parameters **id** (*string*) – the id of the activity

Return type list of dictionaries.

`ckan.logic.action.get.user_activity_list_html` (*context*, *data_dict*)

Return a user's public activity stream as HTML.

The activity stream is rendered as a snippet of HTML meant to be included in an HTML page, i.e. it doesn't have any HTML header or footer.

Parameters

- **id** (*string*) – The id or name of the user.
- **offset** (*int*) – where to start getting activity items from (optional, default: 0)
- **limit** (*int*) – the maximum number of activities to return (optional, default: 31, the default value is configurable via the `ckan.activity_list_limit` setting)

Return type string

`ckan.logic.action.get.package_activity_list_html` (*context*, *data_dict*)

Return a package's activity stream as HTML.

The activity stream is rendered as a snippet of HTML meant to be included in an HTML page, i.e. it doesn't have any HTML header or footer.

Parameters

- **id** (*string*) – the id or name of the package
- **offset** (*int*) – where to start getting activity items from (optional, default: 0)
- **limit** (*int*) – the maximum number of activities to return (optional, default: 31, the default value is configurable via the `ckan.activity_list_limit` setting)

Return type string

`ckan.logic.action.get.group_activity_list_html(context, data_dict)`

Return a group's activity stream as HTML.

The activity stream is rendered as a snippet of HTML meant to be included in an HTML page, i.e. it doesn't have any HTML header or footer.

Parameters

- **id** (*string*) – the id or name of the group
- **offset** (*int*) – where to start getting activity items from (optional, default: 0)
- **limit** (*int*) – the maximum number of activities to return (optional, default: 31, the default value is configurable via the `ckan.activity_list_limit` setting)

Return type string

`ckan.logic.action.get.organization_activity_list_html(context, data_dict)`

Return a organization's activity stream as HTML.

The activity stream is rendered as a snippet of HTML meant to be included in an HTML page, i.e. it doesn't have any HTML header or footer.

Parameters **id** (*string*) – the id or name of the organization

Return type string

`ckan.logic.action.get.recently_changed_packages_activity_list_html(context, data_dict)`

Return the activity stream of all recently changed packages as HTML.

The activity stream includes all recently added or changed packages. It is rendered as a snippet of HTML meant to be included in an HTML page, i.e. it doesn't have any HTML header or footer.

Parameters

- **offset** (*int*) – where to start getting activity items from (optional, default: 0)
- **limit** (*int*) – the maximum number of activities to return (optional, default: 31, the default value is configurable via the `ckan.activity_list_limit` setting)

Return type string

`ckan.logic.action.get.user_follower_count(context, data_dict)`

Return the number of followers of a user.

Parameters **id** (*string*) – the id or name of the user

Return type int

`ckan.logic.action.get.dataset_follower_count(context, data_dict)`

Return the number of followers of a dataset.

Parameters **id** (*string*) – the id or name of the dataset

Return type int

`ckan.logic.action.get.group_follower_count(context, data_dict)`

Return the number of followers of a group.

Parameters **id** (*string*) – the id or name of the group

Return type int

`ckan.logic.action.get.organization_follower_count(context, data_dict)`

Return the number of followers of an organization.

Parameters **id** (*string*) – the id or name of the organization

Return type int

`ckan.logic.action.get.user_follower_list(context, data_dict)`

Return the list of users that are following the given user.

Parameters `id` (*string*) – the id or name of the user

Return type list of dictionaries

`ckan.logic.action.get.dataset_follower_list(context, data_dict)`

Return the list of users that are following the given dataset.

Parameters `id` (*string*) – the id or name of the dataset

Return type list of dictionaries

`ckan.logic.action.get.group_follower_list(context, data_dict)`

Return the list of users that are following the given group.

Parameters `id` (*string*) – the id or name of the group

Return type list of dictionaries

`ckan.logic.action.get.organization_follower_list(context, data_dict)`

Return the list of users that are following the given organization.

Parameters `id` (*string*) – the id or name of the organization

Return type list of dictionaries

`ckan.logic.action.get.am_following_user(context, data_dict)`

Return True if you're following the given user, False if not.

Parameters `id` (*string*) – the id or name of the user

Return type boolean

`ckan.logic.action.get.am_following_dataset(context, data_dict)`

Return True if you're following the given dataset, False if not.

Parameters `id` (*string*) – the id or name of the dataset

Return type boolean

`ckan.logic.action.get.am_following_group(context, data_dict)`

Return True if you're following the given group, False if not.

Parameters `id` (*string*) – the id or name of the group

Return type boolean

`ckan.logic.action.get.followee_count(context, data_dict)`

Return the number of objects that are followed by the given user.

Counts all objects, of any type, that the given user is following (e.g. followed users, followed datasets, followed groups).

Parameters `id` (*string*) – the id of the user

Return type int

`ckan.logic.action.get.user_followee_count(context, data_dict)`

Return the number of users that are followed by the given user.

Parameters `id` (*string*) – the id of the user

Return type int

`ckan.logic.action.get.dataset_followee_count(context, data_dict)`

Return the number of datasets that are followed by the given user.

Parameters `id` (*string*) – the id of the user

Return type `int`

`ckan.logic.action.get.group_followee_count(context, data_dict)`

Return the number of groups that are followed by the given user.

Parameters `id` (*string*) – the id of the user

Return type `int`

`ckan.logic.action.get.followee_list(context, data_dict)`

Return the list of objects that are followed by the given user.

Returns all objects, of any type, that the given user is following (e.g. followed users, followed datasets, followed groups..).

Parameters

- `id` (*string*) – the id of the user
- `q` (*string*) – a query string to limit results by, only objects whose display name begins with the given string (case-insensitive) will be returned (optional)

Return type list of dictionaries, each with keys `'type'` (e.g. `'user'`, `'dataset'` or `'group'`), `'display_name'` (e.g. a user's display name, or a package's title) and `'dict'` (e.g. a dict representing the followed user, package or group, the same as the dict that would be returned by `user_show()`, `package_show()` or `group_show()`)

`ckan.logic.action.get.user_followee_list(context, data_dict)`

Return the list of users that are followed by the given user.

Parameters `id` (*string*) – the id of the user

Return type list of dictionaries

`ckan.logic.action.get.dataset_followee_list(context, data_dict)`

Return the list of datasets that are followed by the given user.

Parameters `id` (*string*) – the id or name of the user

Return type list of dictionaries

`ckan.logic.action.get.group_followee_list(context, data_dict)`

Return the list of groups that are followed by the given user.

Parameters `id` (*string*) – the id or name of the user

Return type list of dictionaries

`ckan.logic.action.get.organization_followee_list(context, data_dict)`

Return the list of organizations that are followed by the given user.

Parameters `id` (*string*) – the id or name of the user

Return type list of dictionaries

`ckan.logic.action.get.dashboard_activity_list(context, data_dict)`

Return the authorized user's dashboard activity stream.

Unlike the activity dictionaries returned by other `*_activity_list` actions, these activity dictionaries have an extra boolean value with key `is_new` that tells you whether the activity happened since the user last viewed her dashboard (`'is_new': True`) or not (`'is_new': False`).

The user's own activities are always marked 'is_new' : False.

Parameters

- **offset** (*int*) – where to start getting activity items from (optional, default: 0)
- **limit** – the maximum number of activities to return (optional, default: 31, the default value is configurable via the `ckan.activity_list_limit` setting)

Return type list of activity dictionaries

`ckan.logic.action.get.dashboard_activity_list_html (context, data_dict)`

Return the authorized user's dashboard activity stream as HTML.

The activity stream is rendered as a snippet of HTML meant to be included in an HTML page, i.e. it doesn't have any HTML header or footer.

Parameters

- **id** (*string*) – the id or name of the user
- **offset** (*int*) – where to start getting activity items from (optional, default: 0)
- **limit** (*int*) – the maximum number of activities to return (optional, default: 31, the default value is configurable via the `ckan.activity_list_limit` setting)

Return type string

`ckan.logic.action.get.dashboard_new_activities_count (context, data_dict)`

Return the number of new activities in the user's dashboard.

Return the number of new activities in the authorized user's dashboard activity stream.

Activities from the user herself are not counted by this function even though they appear in the dashboard (users don't want to be notified about things they did themselves).

Return type int

`ckan.logic.action.get.member_roles_list (context, data_dict)`

Return the possible roles for members of groups and organizations.

Parameters **group_type** – the group type, either "group" or "organization" (optional, default "organization")

Returns a list of dictionaries each with two keys: "text" (the display name of the role, e.g. "Admin") and "value" (the internal name of the role, e.g. "admin")

Return type list of dictionaries

`ckan.logic.action.get.help_show (context, data_dict)`

Return the help string for a particular API action.

Parameters **name** (*string*) – Action function name (eg `user_create`, `package_search`)

Returns The help string for the action function, or None if the function does not have a docstring.

Return type string

Raises `ckan.logic.NotFound`: if the action function doesn't exist

`ckan.logic.action.get.config_option_show (context, data_dict)`

Show the current value of a particular configuration option.

Only returns runtime-editable config options (the ones returned by `config_option_list()`), which can be updated with the `config_option_update()` action.

Parameters **id** (*string*) – The configuration option key

Returns The value of the config option from either the system_info table or ini file.

Return type string

Raises `ckan.logic.ValidationError`: if config option is not in the schema (whitelisted as editable).

`ckan.logic.action.get.config_option_list(context, data_dict)`

Return a list of runtime-editable config options keys that can be updated with
`config_option_update()`.

Returns A list of config option keys.

Return type list

ckan.logic.action.create

API functions for adding data to CKAN.

`ckan.logic.action.create.package_create(context, data_dict)`

Create a new dataset (package).

You must be authorized to create new datasets. If you specify any groups for the new dataset, you must also be authorized to edit these groups.

Plugins may change the parameters of this function depending on the value of the `type` parameter, see the `IDatasetForm` plugin interface.

Parameters

- **name** (*string*) – the name of the new dataset, must be between 2 and 100 characters long and contain only lowercase alphanumeric characters, `-` and `_`, e.g. `'warandpeace'`
- **title** (*string*) – the title of the dataset (optional, default: same as name)
- **author** (*string*) – the name of the dataset's author (optional)
- **author_email** (*string*) – the email address of the dataset's author (optional)
- **maintainer** (*string*) – the name of the dataset's maintainer (optional)
- **maintainer_email** (*string*) – the email address of the dataset's maintainer (optional)
- **license_id** (*license id string*) – the id of the dataset's license, see `license_list()` for available values (optional)
- **notes** (*string*) – a description of the dataset (optional)
- **url** (*string*) – a URL for the dataset's source (optional)
- **version** (*string, no longer than 100 characters*) – (optional)
- **state** (*string*) – the current state of the dataset, e.g. `'active'` or `'deleted'`, only active datasets show up in search results and other lists of datasets, this parameter will be ignored if you are not authorized to change the state of the dataset (optional, default: `'active'`)
- **type** (*string*) – the type of the dataset (optional), `IDatasetForm` plugins associate themselves with different dataset types and provide custom dataset handling behaviour for these types
- **resources** (*list of resource dictionaries*) – the dataset's resources, see `resource_create()` for the format of resource dictionaries (optional)

- **tags** (*list of tag dictionaries*) – the dataset’s tags, see `tag_create()` for the format of tag dictionaries (optional)
- **extras** (*list of dataset extra dictionaries*) – the dataset’s extras (optional), extras are arbitrary (key: value) metadata items that can be added to datasets, each extra dictionary should have keys `'key'` (a string), `'value'` (a string)
- **relationships_as_object** (*list of relationship dictionaries*) – see `package_relationship_create()` for the format of relationship dictionaries (optional)
- **relationships_as_subject** (*list of relationship dictionaries*) – see `package_relationship_create()` for the format of relationship dictionaries (optional)
- **groups** (*list of dictionaries*) – the groups to which the dataset belongs (optional), each group dictionary should have one or more of the following keys which identify an existing group: `'id'` (the id of the group, string), or `'name'` (the name of the group, string), to see which groups exist call `group_list()`
- **owner_org** (*string*) – the id of the dataset’s owning organization, see `organization_list()` or `organization_list_for_user()` for available values (optional)

Returns the newly created dataset (unless `'return_id_only'` is set to True in the context, in which case just the dataset id will be returned)

Return type dictionary

`ckan.logic.action.create.resource_create(context, data_dict)`

Appends a new resource to a datasets list of resources.

Parameters

- **package_id** (*string*) – id of package that the resource should be added to.
- **url** (*string*) – url of resource
- **revision_id** (*string*) – (optional)
- **description** (*string*) – (optional)
- **format** (*string*) – (optional)
- **hash** (*string*) – (optional)
- **name** (*string*) – (optional)
- **resource_type** (*string*) – (optional)
- **mimetype** (*string*) – (optional)
- **mimetype_inner** (*string*) – (optional)
- **webstore_url** (*string*) – (optional)
- **cache_url** (*string*) – (optional)
- **size** (*int*) – (optional)
- **created** (*iso date string*) – (optional)
- **last_modified** (*iso date string*) – (optional)
- **cache_last_updated** (*iso date string*) – (optional)
- **webstore_last_updated** (*iso date string*) – (optional)

- **upload** (*FieldStorage (optional) needs multipart/form-data*) – (optional)

Returns the newly created resource

Return type dictionary

```
ckan.logic.action.create.resource_view_create(context, data_dict)
```

Creates a new resource view.

Parameters

- **resource_id** (*string*) – id of the resource
- **title** (*string*) – the title of the view
- **description** (*string*) – a description of the view (optional)
- **view_type** (*string*) – type of view
- **config** (*JSON string*) – options necessary to recreate a view state (optional)

Returns the newly created resource view

Return type dictionary

```
ckan.logic.action.create.resource_create_default_resource_views(context,  
                                                                data_dict)
```

Creates the default views (if necessary) on the provided resource

The function will get the plugins for the default views defined in the configuration, and if some were found the *can_view* method of each one of them will be called to determine if a resource view should be created. Resource views extensions get the resource dict and the parent dataset dict.

If the latter is not provided, *package_show* is called to get it.

By default only view plugins that don't require the resource data to be in the DataStore are called. See `ckan.logic.action.create.package_create_default_resource_views.``()` for details on the `create_datastore_views` parameter.

Parameters

- **resource** (*dict*) – full resource dict
- **package** (*dict*) – full dataset dict (optional, if not provided `package_show()` will be called).
- **create_datastore_views** (*bool*) – whether to create views that rely on data being on the DataStore (optional, defaults to False)

Returns a list of resource views created (empty if none were created)

Return type list of dictionaries

```
ckan.logic.action.create.package_create_default_resource_views(context,  
                                                                data_dict)
```

Creates the default views on all resources of the provided dataset

By default only view plugins that don't require the resource data to be in the DataStore are called. Passing `create_datastore_views` as True will only create views that require data to be in the DataStore. The first case happens when the function is called from *package_create* or *package_update*, the second when it's called from the DataPusher when data was uploaded to the DataStore.

Parameters

- **package** (*dict*) – full dataset dict (ie the one obtained calling `package_show()`).

- **create_datastore_views** (*bool*) – whether to create views that rely on data being on the DataStore (optional, defaults to False)

Returns a list of resource views created (empty if none were created)

Return type list of dictionaries

`ckan.logic.action.create.related_create(context, data_dict)`

Add a new related item to a dataset.

You must provide your API key in the Authorization header.

Parameters

- **title** (*string*) – the title of the related item
- **type** (*string*) – the type of the related item, e.g. 'Application', 'Idea' or 'Visualisation'
- **id** (*string*) – the id of the related item (optional)
- **description** (*string*) – the description of the related item (optional)
- **url** (*string*) – the URL to the related item (optional)
- **image_url** (*string*) – the URL to the image for the related item (optional)
- **dataset_id** (*string*) – the name or id of the dataset that the related item belongs to (optional)

Returns the newly created related item

Return type dictionary

`ckan.logic.action.create.package_relationship_create(context, data_dict)`

Create a relationship between two datasets (packages).

You must be authorized to edit both the subject and the object datasets.

Parameters

- **subject** (*string*) – the id or name of the dataset that is the subject of the relationship
- **object** – the id or name of the dataset that is the object of the relationship
- **type** (*string*) – the type of the relationship, one of 'depends_on', 'dependency_of', 'derives_from', 'has_derivation', 'links_to', 'linked_from', 'child_of' or 'parent_of'
- **comment** (*string*) – a comment about the relationship (optional)

Returns the newly created package relationship

Return type dictionary

`ckan.logic.action.create.member_create(context, data_dict=None)`

Make an object (e.g. a user, dataset or group) a member of a group.

If the object is already a member of the group then the capacity of the membership will be updated.

You must be authorized to edit the group.

Parameters

- **id** (*string*) – the id or name of the group to add the object to
- **object** (*string*) – the id or name of the object to add
- **object_type** (*string*) – the type of the object being added, e.g. 'package' or 'user'

- **capacity** (*string*) – the capacity of the membership

Returns the newly created (or updated) membership

Return type dictionary

`ckan.logic.action.create.group_create(context, data_dict)`

Create a new group.

You must be authorized to create groups.

Plugins may change the parameters of this function depending on the value of the `type` parameter, see the [IGroupForm](#) plugin interface.

Parameters

- **name** (*string*) – the name of the group, a string between 2 and 100 characters long, containing only lowercase alphanumeric characters, `-` and `_`
- **id** (*string*) – the id of the group (optional)
- **title** (*string*) – the title of the group (optional)
- **description** (*string*) – the description of the group (optional)
- **image_url** (*string*) – the URL to an image to be displayed on the group's page (optional)
- **type** (*string*) – the type of the group (optional), [IGroupForm](#) plugins associate themselves with different group types and provide custom group handling behaviour for these types Cannot be 'organization'
- **state** (*string*) – the current state of the group, e.g. 'active' or 'deleted', only active groups show up in search results and other lists of groups, this parameter will be ignored if you are not authorized to change the state of the group (optional, default: 'active')
- **approval_status** (*string*) – (optional)
- **extras** (*list of dataset extra dictionaries*) – the group's extras (optional), extras are arbitrary (key: value) metadata items that can be added to groups, each extra dictionary should have keys 'key' (a string), 'value' (a string), and optionally 'deleted'
- **packages** (*list of dictionaries*) – the datasets (packages) that belong to the group, a list of dictionaries each with keys 'name' (string, the id or name of the dataset) and optionally 'title' (string, the title of the dataset)
- **groups** (*list of dictionaries*) – the groups that belong to the group, a list of dictionaries each with key 'name' (string, the id or name of the group) and optionally 'capacity' (string, the capacity in which the group is a member of the group)
- **users** (*list of dictionaries*) – the users that belong to the group, a list of dictionaries each with key 'name' (string, the id or name of the user) and optionally 'capacity' (string, the capacity in which the user is a member of the group)

Returns the newly created group (unless 'return_id_only' is set to True in the context, in which case just the group id will be returned)

Return type dictionary

`ckan.logic.action.create.organization_create(context, data_dict)`

Create a new organization.

You must be authorized to create organizations.

Plugins may change the parameters of this function depending on the value of the `type` parameter, see the [IGroupForm](#) plugin interface.

Parameters

- **name** (*string*) – the name of the organization, a string between 2 and 100 characters long, containing only lowercase alphanumeric characters, – and _
- **id** (*string*) – the id of the organization (optional)
- **title** (*string*) – the title of the organization (optional)
- **description** (*string*) – the description of the organization (optional)
- **image_url** (*string*) – the URL to an image to be displayed on the organization's page (optional)
- **state** (*string*) – the current state of the organization, e.g. 'active' or 'deleted', only active organizations show up in search results and other lists of organizations, this parameter will be ignored if you are not authorized to change the state of the organization (optional, default: 'active')
- **approval_status** (*string*) – (optional)
- **extras** (*list of dataset extra dictionaries*) – the organization's extras (optional), extras are arbitrary (key: value) metadata items that can be added to organizations, each extra dictionary should have keys 'key' (a string), 'value' (a string), and optionally 'deleted'
- **packages** (*list of dictionaries*) – the datasets (packages) that belong to the organization, a list of dictionaries each with keys 'name' (string, the id or name of the dataset) and optionally 'title' (string, the title of the dataset)
- **users** (*list of dictionaries*) – the users that belong to the organization, a list of dictionaries each with key 'name' (string, the id or name of the user) and optionally 'capacity' (string, the capacity in which the user is a member of the organization)

Returns the newly created organization (unless 'return_id_only' is set to True in the context, in which case just the organization id will be returned)

Return type dictionary

```
ckan.logic.action.create.rating_create(context, data_dict)
```

Rate a dataset (package).

You must provide your API key in the Authorization header.

Parameters

- **package** (*string*) – the name or id of the dataset to rate
- **rating** (*int*) – the rating to give to the dataset, an integer between 1 and 5

Returns a dictionary with two keys: 'rating_average' (the average rating of the dataset you rated) and 'rating_count' (the number of times the dataset has been rated)

Return type dictionary

```
ckan.logic.action.create.user_create(context, data_dict)
```

Create a new user.

You must be authorized to create users.

Parameters

- **name** (*string*) – the name of the new user, a string between 2 and 100 characters in length, containing only lowercase alphanumeric characters, – and _
- **email** (*string*) – the email address for the new user

- **password** (*string*) – the password of the new user, a string of at least 4 characters
- **id** (*string*) – the id of the new user (optional)
- **fullname** (*string*) – the full name of the new user (optional)
- **about** (*string*) – a description of the new user (optional)
- **openid** (*string*) – (optional)

Returns the newly created user

Return type dictionary

`ckan.logic.action.create.user_invite(context, data_dict)`

Invite a new user.

You must be authorized to create group members.

Parameters

- **email** (*string*) – the email of the user to be invited to the group
- **group_id** (*string*) – the id or name of the group
- **role** (*string*) – role of the user in the group. One of member, editor, or admin

Returns the newly created user

Return type dictionary

`ckan.logic.action.create.vocabulary_create(context, data_dict)`

Create a new tag vocabulary.

You must be a sysadmin to create vocabularies.

Parameters

- **name** (*string*) – the name of the new vocabulary, e.g. 'Genre'
- **tags** (*list of tag dictionaries*) – the new tags to add to the new vocabulary, for the format of tag dictionaries see [tag_create\(\)](#)

Returns the newly-created vocabulary

Return type dictionary

`ckan.logic.action.create.activity_create(context, activity_dict, **kw)`

Create a new activity stream activity.

You must be a sysadmin to create new activities.

Parameters

- **user_id** (*string*) – the name or id of the user who carried out the activity, e.g. 'seanh'
- **object_id** – the name or id of the object of the activity, e.g. 'my_dataset'
- **activity_type** (*string*) – the type of the activity, this must be an activity type that CKAN knows how to render, e.g. 'new package', 'changed user', 'deleted group' etc.
- **data** (*dictionary*) – any additional data about the activity

Returns the newly created activity

Return type dictionary

`ckan.logic.action.create.tag_create(context, data_dict)`

Create a new vocabulary tag.

You must be a sysadmin to create vocabulary tags.

You can only use this function to create tags that belong to a vocabulary, not to create free tags. (To create a new free tag simply add the tag to a package, e.g. using the `package_update()` function.)

Parameters

- **name** (*string*) – the name for the new tag, a string between 2 and 100 characters long containing only alphanumeric characters and `-`, `_` and `.`, e.g. `'Jazz'`
- **vocabulary_id** (*string*) – the id of the vocabulary that the new tag should be added to, e.g. the id of vocabulary `'Genre'`

Returns the newly-created tag

Return type dictionary

`ckan.logic.action.create.follow_user(context, data_dict)`

Start following another user.

You must provide your API key in the Authorization header.

Parameters **id** (*string*) – the id or name of the user to follow, e.g. `'joeuser'`

Returns a representation of the ‘follower’ relationship between yourself and the other user

Return type dictionary

`ckan.logic.action.create.follow_dataset(context, data_dict)`

Start following a dataset.

You must provide your API key in the Authorization header.

Parameters **id** (*string*) – the id or name of the dataset to follow, e.g. `'warandpeace'`

Returns a representation of the ‘follower’ relationship between yourself and the dataset

Return type dictionary

`ckan.logic.action.create.group_member_create(context, data_dict)`

Make a user a member of a group.

You must be authorized to edit the group.

Parameters

- **id** (*string*) – the id or name of the group
- **username** (*string*) – name or id of the user to be made member of the group
- **role** (*string*) – role of the user in the group. One of `member`, `editor`, or `admin`

Returns the newly created (or updated) membership

Return type dictionary

`ckan.logic.action.create.organization_member_create(context, data_dict)`

Make a user a member of an organization.

You must be authorized to edit the organization.

Parameters

- **id** (*string*) – the id or name of the organization
- **username** (*string*) – name or id of the user to be made member of the organization

- **role** (*string*) – role of the user in the organization. One of member, editor, or admin

Returns the newly created (or updated) membership

Return type dictionary

`ckan.logic.action.create.follow_group(context, data_dict)`

Start following a group.

You must provide your API key in the Authorization header.

Parameters **id** (*string*) – the id or name of the group to follow, e.g. 'roger'

Returns a representation of the 'follower' relationship between yourself and the group

Return type dictionary

ckan.logic.action.update

API functions for updating existing data in CKAN.

`ckan.logic.action.update.related_update(context, data_dict)`

Update a related item.

You must be the owner of a related item to update it.

For further parameters see `related_create()`.

Parameters **id** (*string*) – the id of the related item to update

Returns the updated related item

Return type dictionary

`ckan.logic.action.update.resource_update(context, data_dict)`

Update a resource.

To update a resource you must be authorized to update the dataset that the resource belongs to.

For further parameters see `resource_create()`.

Parameters **id** (*string*) – the id of the resource to update

Returns the updated resource

Return type string

`ckan.logic.action.update.resource_view_update(context, data_dict)`

Update a resource view.

To update a resource_view you must be authorized to update the resource that the resource_view belongs to.

For further parameters see `resource_view_create()`.

Parameters **id** (*string*) – the id of the resource_view to update

Returns the updated resource_view

Return type string

`ckan.logic.action.update.resource_view_reorder(context, data_dict)`

Reorder resource views.

Parameters

- **id** (*string*) – the id of the resource
- **order** (*list of strings*) – the list of id of the resource to update the order of the views

Returns the updated order of the view

Return type dictionary

`ckan.logic.action.update.package_update(context, data_dict)`

Update a dataset (package).

You must be authorized to edit the dataset and the groups that it belongs to.

It is recommended to call `ckan.logic.action.get.package_show()`, make the desired changes to the result, and then call `package_update()` with it.

Plugins may change the parameters of this function depending on the value of the dataset's `type` attribute, see the `IDatasetForm` plugin interface.

For further parameters see `package_create()`.

Parameters `id` (*string*) – the name or id of the dataset to update

Returns the updated dataset (if `'return_package_dict'` is `True` in the context, which is the default. Otherwise returns just the dataset id)

Return type dictionary

`ckan.logic.action.update.package_resource_reorder(context, data_dict)`

Reorder resources against datasets. If only partial resource ids are supplied then these are assumed to be first and the other resources will stay in their original order

Parameters

- **id** (*string*) – the id or name of the package to update
- **order** – a list of resource ids in the order needed

`ckan.logic.action.update.package_relationship_update(context, data_dict)`

Update a relationship between two datasets (packages).

You must be authorized to edit both the subject and the object datasets.

Parameters

- **id** (*string*) – the id of the package relationship to update
- **subject** (*string*) – the name or id of the dataset that is the subject of the relationship (optional)
- **object** – the name or id of the dataset that is the object of the relationship (optional)
- **type** (*string*) – the type of the relationship, one of `'depends_on'`, `'dependency_of'`, `'derives_from'`, `'has_derivation'`, `'links_to'`, `'linked_from'`, `'child_of'` or `'parent_of'` (optional)
- **comment** (*string*) – a comment about the relationship (optional)

Returns the updated relationship

Return type dictionary

`ckan.logic.action.update.group_update(context, data_dict)`

Update a group.

You must be authorized to edit the group.

Plugins may change the parameters of this function depending on the value of the group's `type` attribute, see the `IGroupForm` plugin interface.

For further parameters see `group_create()`.

Parameters `id` (*string*) – the name or id of the group to update

Returns the updated group

Return type dictionary

`ckan.logic.action.update.organization_update(context, data_dict)`

Update a organization.

You must be authorized to edit the organization.

For further parameters see `organization_create()`.

Parameters

- `id` (*string*) – the name or id of the organization to update
- `packages` – ignored. use `package_owner_org_update()` to change package ownership

Returns the updated organization

Return type dictionary

`ckan.logic.action.update.user_update(context, data_dict)`

Update a user account.

Normal users can only update their own user accounts. Sysadmins can update any user account.

For further parameters see `user_create()`.

Parameters `id` (*string*) – the name or id of the user to update

Returns the updated user account

Return type dictionary

`ckan.logic.action.update.user_generate_apikey(context, data_dict)`

Cycle a user's API key

Parameters `id` (*string*) – the name or id of the user whose key needs to be updated

Returns the updated user

Return type dictionary

`ckan.logic.action.update.task_status_update(context, data_dict)`

Update a task status.

Parameters

- `id` (*string*) – the id of the task status to update
- `entity_id` (*string*) –
- `entity_type` (*string*) –
- `task_type` (*string*) –
- `key` (*string*) –
- `value` – (optional)
- `state` – (optional)
- `last_updated` – (optional)
- `error` – (optional)

Returns the updated task status

Return type dictionary

`ckan.logic.action.update.task_status_update_many(context, data_dict)`

Update many task statuses at once.

Parameters **data** (*list of dictionaries*) – the task_status dictionaries to update, for the format of task status dictionaries see `task_status_update()`

Returns the updated task statuses

Return type list of dictionaries

`ckan.logic.action.update.term_translation_update(context, data_dict)`

Create or update a term translation.

You must be a sysadmin to create or update term translations.

Parameters

- **term** (*string*) – the term to be translated, in the original language, e.g. 'romantic novel'
- **term_translation** (*string*) – the translation of the term, e.g. 'Liebesroman'
- **lang_code** (*string*) – the language code of the translation, e.g. 'de'

Returns the newly created or updated term translation

Return type dictionary

`ckan.logic.action.update.term_translation_update_many(context, data_dict)`

Create or update many term translations at once.

Parameters **data** (*list of dictionaries*) – the term translation dictionaries to create or update, for the format of term translation dictionaries see `term_translation_update()`

Returns a dictionary with key 'success' whose value is a string stating how many term translations were updated

Return type string

`ckan.logic.action.update.vocabulary_update(context, data_dict)`

Update a tag vocabulary.

You must be a sysadmin to update vocabularies.

For further parameters see `vocabulary_create()`.

Parameters **id** (*string*) – the id of the vocabulary to update

Returns the updated vocabulary

Return type dictionary

`ckan.logic.action.update.user_role_update(context, data_dict)`

Update a user or authorization group's roles for a domain object.

The `user` parameter must be given.

You must be authorized to update the domain object.

To delete all of a user or authorization group's roles for domain object, pass an empty list [] to the `roles` parameter.

Parameters

- **user** (*string*) – the name or id of the user

- **domain_object** (*string*) – the name or id of the domain object (e.g. a package, group or authorization group)
- **roles** (*list of strings*) – the new roles, e.g. ['editor']

Returns the updated roles of all users for the domain object

Return type dictionary

`ckan.logic.action.update.user_role_bulk_update(context, data_dict)`

Update the roles of many users or authorization groups for an object.

You must be authorized to update the domain object.

Parameters **user_roles** (*list of dictionaries*) – the updated user roles, for the format of user role dictionaries see `user_role_update()`

Returns the updated roles of all users and authorization groups for the domain object

Return type dictionary

`ckan.logic.action.update.dashboard_mark_activities_old(context, data_dict)`

Mark all the authorized user's new dashboard activities as old.

This will reset `dashboard_new_activities_count()` to 0.

`ckan.logic.action.update.send_email_notifications(context, data_dict)`

Send any pending activity stream notification emails to users.

You must provide a sysadmin's API key in the Authorization header of the request, or call this action from the command-line via a *paster post ...* command.

`ckan.logic.action.update.package_owner_org_update(context, data_dict)`

Update the owning organization of a dataset

Parameters

- **id** (*string*) – the name or id of the dataset to update
- **organization_id** – the name or id of the owning organization

`ckan.logic.action.update.bulk_update_private(context, data_dict)`

Make a list of datasets private

Parameters

- **datasets** (*list of strings*) – list of ids of the datasets to update
- **org_id** (*int*) – id of the owning organization

`ckan.logic.action.update.bulk_update_public(context, data_dict)`

Make a list of datasets public

Parameters

- **datasets** (*list of strings*) – list of ids of the datasets to update
- **org_id** (*int*) – id of the owning organization

`ckan.logic.action.update.bulk_update_delete(context, data_dict)`

Make a list of datasets deleted

Parameters

- **datasets** (*list of strings*) – list of ids of the datasets to update
- **org_id** (*int*) – id of the owning organization

`ckan.logic.action.update.config_option_update(context, data_dict)`

New in version 2.4.

Allows to modify some CKAN runtime-editable config options

It takes arbitrary key, value pairs and checks the keys against the config options update schema. If some of the provided keys are not present in the schema a `ValidationError` is raised. The values are then validated against the schema, and if validation is passed, for each key, value config option:

- It is stored on the `system_info` database table
- The Pylons `config` object is updated.
- The `app_globals (g)` object is updated (this only happens for options explicitly defined in the `app_globals` module).

The following lists a key parameter, but this should be replaced by whichever config options want to be updated, eg:

```
get_action('config_option_update')({}, {
    'ckan.site_title': 'My Open Data site',
    'ckan.homepage_layout': 2,
})
```

Parameters `key (string)` – a configuration option key (eg `ckan.site_title`). It must be present on the `update_configuration_schema`

Returns a dictionary with the options set

Return type dictionary

Note: You can see all available runtime-editable configuration options calling the `config_option_list()` action

Note: Extensions can modify which configuration options are runtime-editable. For details, check [Making configuration options runtime-editable](#).

Warning: You should only add config options that you are comfortable they can be edited during runtime, such as ones you’ve added in your own extension, or have reviewed the use of in core CKAN.

ckan.logic.action.patch

New in version 2.3. API functions for partial updates of existing data in CKAN

`ckan.logic.action.patch.package_patch(context, data_dict)`

Patch a dataset (package).

Parameters `id (string)` – the id or name of the dataset

The difference between the update and patch methods is that the patch will perform an update of the provided parameters, while leaving all other parameters unchanged, whereas the update methods deletes all parameters not explicitly provided in the `data_dict`

You must be authorized to edit the dataset and the groups that it belongs to.

`ckan.logic.action.patch.resource_patch(context, data_dict)`

Patch a resource

Parameters **id** (*string*) – the id of the resource

The difference between the update and patch methods is that the patch will perform an update of the provided parameters, while leaving all other parameters unchanged, whereas the update methods deletes all parameters not explicitly provided in the data_dict

```
ckan.logic.action.patch.group_patch (context, data_dict)
```

Patch a group

Parameters **id** (*string*) – the id or name of the group

The difference between the update and patch methods is that the patch will perform an update of the provided parameters, while leaving all other parameters unchanged, whereas the update methods deletes all parameters not explicitly provided in the data_dict

```
ckan.logic.action.patch.organization_patch (context, data_dict)
```

Patch an organization

Parameters **id** (*string*) – the id or name of the organization

The difference between the update and patch methods is that the patch will perform an update of the provided parameters, while leaving all other parameters unchanged, whereas the update methods deletes all parameters not explicitly provided in the data_dict

ckan.logic.action.delete

API functions for deleting data from CKAN.

```
ckan.logic.action.delete.user_delete (context, data_dict)
```

Delete a user.

Only sysadmins can delete users.

Parameters **id** (*string*) – the id or username of the user to delete

```
ckan.logic.action.delete.package_delete (context, data_dict)
```

Delete a dataset (package).

You must be authorized to delete the dataset.

Parameters **id** (*string*) – the id or name of the dataset to delete

```
ckan.logic.action.delete.resource_delete (context, data_dict)
```

Delete a resource from a dataset.

You must be a sysadmin or the owner of the resource to delete it.

Parameters **id** (*string*) – the id of the resource

```
ckan.logic.action.delete.resource_view_delete (context, data_dict)
```

Delete a resource_view.

Parameters **id** (*string*) – the id of the resource_view

```
ckan.logic.action.delete.resource_view_clear (context, data_dict)
```

Delete all resource views, or all of a particular type.

Parameters **view_types** (*list*) – specific types to delete (optional)

```
ckan.logic.action.delete.package_relationship_delete (context, data_dict)
```

Delete a dataset (package) relationship.

You must be authorised to delete dataset relationships, and to edit both the subject and the object datasets.

Parameters

- **subject** (*string*) – the id or name of the dataset that is the subject of the relationship
- **object** (*string*) – the id or name of the dataset that is the object of the relationship
- **type** (*string*) – the type of the relationship

`ckan.logic.action.delete.related_delete(context, data_dict)`

Delete a related item from a dataset.

You must be a sysadmin or the owner of the related item to delete it.

Parameters **id** (*string*) – the id of the related item

`ckan.logic.action.delete.member_delete(context, data_dict=None)`

Remove an object (e.g. a user, dataset or group) from a group.

You must be authorized to edit a group to remove objects from it.

Parameters

- **id** (*string*) – the id of the group
- **object** (*string*) – the id or name of the object to be removed
- **object_type** (*string*) – the type of the object to be removed, e.g. package or user

`ckan.logic.action.delete.group_delete(context, data_dict)`

Delete a group.

You must be authorized to delete the group.

Parameters **id** (*string*) – the name or id of the group

`ckan.logic.action.delete.organization_delete(context, data_dict)`

Delete an organization.

You must be authorized to delete the organization.

Parameters **id** (*string*) – the name or id of the organization

`ckan.logic.action.delete.group_purge(context, data_dict)`

Purge a group.

Warning: Purging a group cannot be undone!

Purging a group completely removes the group from the CKAN database, whereas deleting a group simply marks the group as deleted (it will no longer show up in the frontend, but is still in the db).

You must be authorized to purge the group.

Parameters **id** (*string*) – the name or id of the group to be purged

`ckan.logic.action.delete.organization_purge(context, data_dict)`

Purge an organization.

Warning: Purging an organization cannot be undone!

Purging an organization completely removes the organization from the CKAN database, whereas deleting an organization simply marks the organization as deleted (it will no longer show up in the frontend, but is still in the db).

You must be authorized to purge the organization.

Parameters **id** (*string*) – the name or id of the organization to be purged

`ckan.logic.action.delete.task_status_delete(context, data_dict)`

Delete a task status.

You must be a sysadmin to delete task statuses.

Parameters `id` (*string*) – the id of the task status to delete

`ckan.logic.action.delete.vocabulary_delete(context, data_dict)`

Delete a tag vocabulary.

You must be a sysadmin to delete vocabularies.

Parameters `id` (*string*) – the id of the vocabulary

`ckan.logic.action.delete.tag_delete(context, data_dict)`

Delete a tag.

You must be a sysadmin to delete tags.

Parameters

- `id` (*string*) – the id or name of the tag
- `vocabulary_id` (*string*) – the id or name of the vocabulary that the tag belongs to (optional, default: None)

`ckan.logic.action.delete.unfollow_user(context, data_dict)`

Stop following a user.

Parameters `id` (*string*) – the id or name of the user to stop following

`ckan.logic.action.delete.unfollow_dataset(context, data_dict)`

Stop following a dataset.

Parameters `id` (*string*) – the id or name of the dataset to stop following

`ckan.logic.action.delete.group_member_delete(context, data_dict=None)`

Remove a user from a group.

You must be authorized to edit the group.

Parameters

- `id` (*string*) – the id or name of the group
- `username` (*string*) – name or id of the user to be removed

`ckan.logic.action.delete.organization_member_delete(context, data_dict=None)`

Remove a user from an organization.

You must be authorized to edit the organization.

Parameters

- `id` (*string*) – the id or name of the organization
- `username` (*string*) – name or id of the user to be removed

`ckan.logic.action.delete.unfollow_group(context, data_dict)`

Stop following a group.

Parameters `id` (*string*) – the id or name of the group to stop following

Extending guide

The following sections will teach you how to customize and extend CKAN's features by developing your own CKAN extensions.

See also:

Some **core extensions** come packaged with CKAN. Core extensions don't need to be installed before you can use them as they're installed when you install CKAN, they can simply be enabled by following the setup instructions in each extension's documentation (some core extensions are already enabled by default). For example, the *datastore extension*, *multilingual extension*, and *stats extension* are all core extensions, and the *data viewer* also uses core extensions to enable data previews for different file formats.

See also:

External extensions are CKAN extensions that don't come packaged with CKAN, but must be downloaded and installed separately. A good place to find external extensions is the [list of extensions on the CKAN wiki](#). Again, follow each extension's own documentation to install, setup and use the extension.

Writing extensions tutorial

This tutorial will walk you through the process of creating a simple CKAN extension, and introduce the core concepts that CKAN extension developers need to know along the way. As an example, we'll use the *example_iauthfunctions* extension that's packaged with CKAN. This is a simple CKAN extension that customizes some of CKAN's authorization rules.

Installing CKAN

Before you can start developing a CKAN extension, you'll need a working source install of CKAN on your system. If you don't have a CKAN source install already, follow the instructions in *Installing CKAN from source* before continuing.

Creating a new extension

Extensions

A CKAN *extension* is a Python package that modifies or extends CKAN. Each extension contains one or more *plugins* that must be added to your CKAN config file to activate the extension's features.

You can use the `paster create` command to create an “empty” extension from a template. First, activate your CKAN virtual environment:

```
. /usr/lib/ckan/default/bin/activate
```

When you run the `paster create` command, your new extension’s directory will be created in the current working directory by default (you can override this with the `-o` option), so change to the directory that you want your extension to be created in. Usually you’ll want to track your extension code using a version control system such as `git`, so you wouldn’t want to create your extension in the `ckan` source directory because that directory already contains the CKAN `git` repo. Let’s use the parent directory instead:

```
cd /usr/lib/ckan/default/src
```

Now run the `paster create` command to create your extension:

```
paster --plugin=ckan create -t ckanext ckanext-iauthfunctions
```

Note: The last argument to the `paster create` command (`ckanext-iauthfunctions` in this example) is the name for your next extension. CKAN extension names *have* to begin with `ckanext-`.

The command will ask you to answer a few questions. The answers you give will end up in your extension’s `setup.py` file (where you can edit them later if you want).

Once this command has completed, your new CKAN extension’s project directory will have been created and will contain a few directories and files to get you started:

```
ckanext-iauthfunctions/  
  ckanext/  
    __init__.py  
    iauthfunctions/  
      __init__.py  
  ckanext_iauthfunctions.egg-info/  
  setup.py
```

`ckanext_iauthfunctions.egg_info` is a directory containing automatically generated metadata about your project. It’s used by Python’s packaging and distribution tools. In general, you don’t need to edit or look at anything in this directory, and you should not add it to version control.

`setup.py` is the setup script for your project. As you’ll see later, you use this script to install your project into a virtual environment. It contains several settings that you’ll update as you develop your project.

`ckanext/iauthfunctions` is the Python package directory where we’ll add the source code files for our extension.

Creating a plugin class

Plugins

Each CKAN extension contains one or more plugins that provide the extension’s features.

Now create the file `ckanext-iauthfunctions/ckanext/iauthfunctions/plugin.py` with the following contents:

```
import ckan.plugins as plugins
```

```
class ExampleIAuthFunctionsPlugin(plugins.SingletonPlugin):
    pass
```

Our plugin is a normal Python class, named `ExampleIAuthFunctionsPlugin` in this example, that inherits from CKAN's `SingletonPlugin` class.

Note: Every CKAN plugin class should inherit from `SingletonPlugin`.

Adding the plugin to `setup.py`

Now let's add our class to the `entry_points` in `setup.py`. This identifies the plugin class to CKAN once the extension is installed in CKAN's virtualenv, and associates a plugin name with the class. Edit `ckanext-iauthfunctions/setup.py` and add a line to the `entry_points` section like this:

```
entry_points='''
    [ckan.plugins]
    example_iauthfunctions=ckanext.iauthfunctions.plugin:ExampleIAuthFunctionsPlugin
''',
```

Installing the extension

When you *install CKAN*, you create a Python *virtual environment* in a directory on your system (`/usr/lib/ckan/default` by default) and install the CKAN Python package and the other packages that CKAN depends on into this virtual environment. Before we can use our plugin, we must install our extension into our CKAN virtual environment.

Make sure your virtualenv is activated, change to the extension's directory, and run `python setup.py develop`:

```
. /usr/lib/ckan/default/bin/activate
cd /usr/lib/ckan/default/src/ckanext-iauthfunctions
python setup.py develop
```

Enabling the plugin

An extension's plugins must be added to the *ckan.plugins* setting in your CKAN config file so that CKAN will call the plugins' methods. The name that you gave to your plugin class in the *left-hand-side of the assignment in the setup.py file* (`example_iauthfunctions` in this example) is the name you'll use for your plugin in CKAN's config file:

```
ckan.plugins = stats text_view recline_view example_iauthfunctions
```

You should now be able to start CKAN in the development web server and have it start up without any problems:

```
$ paster serve /etc/ckan/default/development.ini
Starting server in PID 13961.
serving on 0.0.0.0:5000 view at http://127.0.0.1:5000
```

If your plugin is in the *ckan.plugins* setting and CKAN starts without crashing, then your plugin is installed and CKAN can find it. Of course, your plugin doesn't *do* anything yet.

Troubleshooting

PluginNotFound exception

If CKAN crashes with a `PluginNotFound exception` like this:

```
ckan.plugins.core.PluginNotFoundException: example_iauthfunctions
```

then:

- Check that the name you've used for your plugin in your CKAN config file is the same as the name you've used in your extension's `setup.py` file
- Check that you've run `python setup.py develop` in your extension's directory, with your CKAN virtual environment activated. Every time you add a new plugin to your extension's `setup.py` file, you need to run `python setup.py develop` again before you can use the new plugin.

ImportError

If you get an `ImportError` from CKAN relating to your plugin, it's probably because the path to your plugin class in your `setup.py` file is wrong.

Implementing the `IAuthFunctions` plugin interface

Plugin interfaces

CKAN provides a number of *plugin interfaces* that plugins must implement to hook into CKAN and modify or extend it. Each plugin interface defines a number of methods that a plugin that implements the interface must provide. CKAN will call your plugin's implementations of these methods, to allow your plugin to do its stuff.

To modify CKAN's authorization behavior, we'll implement the `IAuthFunctions` plugin interface. This interface defines just one method, that takes no parameters and returns a dictionary:

```
get_auth_functions()    Return the authorization functions provided by this plugin.
```

Action functions and authorization functions

At this point, it's necessary to take a short diversion to explain how authorization works in CKAN.

Every action that can be carried out using the CKAN web interface or API is implemented by an *action function* in one of the four files `ckan/logic/action/{create, delete, get, update}.py`.

For example, when creating a dataset either using the web interface or using the `package_create()` API call, `ckan.logic.action.create.package_create()` is called. There's also `ckan.logic.action.get.package_show()`, `ckan.logic.action.update.package_update()`, and `ckan.logic.action.delete.package_delete()`. For a full list of the action functions available in CKAN, see the [Action API reference](#).

Each action function has a corresponding authorization function in one of the four files `ckan/logic/auth/{create, delete, get, update}.py`, CKAN calls this authorization function to decide whether the user is authorized to carry out the requested action. For example, when creating a new package using the web interface or API, `ckan.logic.auth.create.package_create()` is called.

The `IAuthFunctions` plugin interface allows CKAN plugins to hook into this authorization system to add their own authorization functions or override the default authorization functions. In this way, plugins have complete control to customize CKAN's auth.

Whenever a user tries to create a new group via the web interface or the API, CKAN calls the `group_create()` authorization function to decide whether to allow the action. Let's override this function and simply prevent anyone from creating new groups. Edit your `plugin.py` file so that it looks like this:

```
import ckan.plugins as plugins

def group_create(context, data_dict=None):
    return {'success': False, 'msg': 'No one is allowed to create groups'}

class ExampleIAuthFunctionsPlugin(plugins.SingletonPlugin):
    plugins.implements(plugins.IAuthFunctions)

    def get_auth_functions(self):
        return {'group_create': group_create}
```

Our `ExampleIAuthFunctionsPlugin` class now calls `implements()` to tell CKAN that it implements the `IAuthFunctions` interface, and provides an implementation of the interface's `get_auth_functions()` method that overrides the default `group_create()` function with a custom one. This custom function simply returns `{ 'success': False }` to refuse to let anyone create a new group.

If you now restart CKAN and reload the `/group` page, as long as you're not a sysadmin user you should see the Add Group button disappear. The CKAN web interface automatically hides buttons that the user is not authorized to use. Visiting `/group/new` directly will redirect you to the login page. If you try to call `group_create()` via the API, you'll receive an `Authorization Error` from CKAN:

```
$ http 127.0.0.1:5000/api/3/action/group_create Authorization:*** name=my_group
HTTP/1.0 403 Forbidden
Access-Control-Allow-Headers: X-CKAN-API-KEY, Authorization, Content-Type
Access-Control-Allow-Methods: POST, PUT, GET, DELETE, OPTIONS
Access-Control-Allow-Origin: *
Cache-Control: no-cache
Content-Length: 2866
Content-Type: application/json;charset=utf-8
Date: Wed, 12 Jun 2013 13:38:01 GMT
Pragma: no-cache
Server: PasteWSGIServer/0.5 Python/2.7.4
```

```
{
  "error": {
    "__type": "Authorization Error",
    "message": "Access denied"
  },
  "help": "Create a new group...",
  "success": false
}
```

If you're logged in as a sysadmin user however, you'll still be able to create new groups. Sysadmin users can always carry out any action, they bypass the authorization functions.

Using the plugins toolkit

Let's make our custom authorization function a little smarter, and allow only users who are members of a particular group named `curators` to create new groups.

First run CKAN, login and then create a new group called `curators`. Then edit `plugin.py` so that it looks like this:

Note: This version of `plugin.py` will crash if the user is not logged in or if the site doesn't have a group called

curators. You'll want to create a `curators` group in your CKAN before editing your plugin to look like this. See *Exception handling* below.

```
import ckan.plugins as plugins
import ckan.plugins.toolkit as toolkit

def group_create(context, data_dict=None):

    # Get the user name of the logged-in user.
    user_name = context['user']

    # Get a list of the members of the 'curators' group.
    members = toolkit.get_action('member_list')(
        data_dict={'id': 'curators', 'object_type': 'user'})

    # 'members' is a list of (user_id, object_type, capacity) tuples, we're
    # only interested in the user_ids.
    member_ids = [member_tuple[0] for member_tuple in members]

    # We have the logged-in user's user name, get their user id.
    convert_user_name_or_id_to_id = toolkit.get_converter(
        'convert_user_name_or_id_to_id')
    user_id = convert_user_name_or_id_to_id(user_name, context)

    # Finally, we can test whether the user is a member of the curators group.
    if user_id in member_ids:
        return {'success': True}
    else:
        return {'success': False,
            'msg': 'Only curators are allowed to create groups'}
```

```
class ExampleIAuthFunctionsPlugin(plugins.SingletonPlugin):
    plugins.implements(plugins.IAuthFunctions)

    def get_auth_functions(self):
        return {'group_create': group_create}
```

context

The `context` parameter of our `group_create()` function is a dictionary that CKAN passes to all authorization and action functions containing some computed variables. Our function gets the name of the logged-in user from `context`:

```
user_name = context['user']
```

data_dict

The `data_dict` parameter of our `group_create()` function is another dictionary that CKAN passes to all authorization and action functions. `data_dict` contains any data posted by the user to CKAN, eg. any fields they've completed in a web form they're submitting or any JSON fields they've posted to the API. If we inspect the contents of the `data_dict` passed to our `group_create()` authorization function, we'll see that it contains the details of the group the user wants to create:


```
{'description': u'A really cool group',
 'image_url': u'',
 'name': u'my_group',
 'title': u'My Group',
 'type': 'group',
 'users': [{ 'capacity': 'admin', 'name': u'seanh' }]}
```

The plugins toolkit

CKAN's *plugins toolkit* is a Python module containing core CKAN functions, classes and exceptions for use by CKAN extensions.

The toolkit's `get_action()` function returns a CKAN action function. The action functions available to extensions are the same functions that CKAN uses internally to carry out actions when users make requests to the web interface or API. Our code uses `get_action()` to get the `member_list()` action function, which it uses to get a list of the members of the `curators` group:

```
members = toolkit.get_action('member_list')(
    data_dict={'id': 'curators', 'object_type': 'user'})
```

Calling `member_list()` in this way is equivalent to posting the same data dict to the `/api/3/action/member_list` API endpoint. For other action functions available from `get_action()`, see *Action API reference*.

The toolkit's `get_validator()` function returns validator and converter functions from `ckan.logic.converters` for plugins to use. This is the same set of converter functions that CKAN's action functions use to convert user-provided data. Our code uses `get_validator()` to get the `convert_user_name_or_id_to_id()` converter function, which it uses to convert the name of the logged-in user to their user id:

```
convert_user_name_or_id_to_id = toolkit.get_converter(
    'convert_user_name_or_id_to_id')
user_id = convert_user_name_or_id_to_id(user_name, context)
```

Finally, we can test whether the logged-in user is a member of the `curators` group, and allow or refuse the action:

```
if user_id in member_ids:
    return {'success': True}
else:
    return {'success': False,
           'msg': 'Only curators are allowed to create groups'}
```

Exception handling

There are two bugs in our `plugin.py` file that need to be fixed using exception handling. First, the class will crash if the site does not have a group named `curators`.

Tip: If you've already created a `curators` group and want to test what happens when the site has no `curators` group, you can use CKAN's command line interface to *clean and reinitialize your database*.

Try visiting the `/group` page in CKAN with our `example_iauthfunctions` plugin activated in your CKAN config file and with no `curators` group in your site. If you have `debug = false` in your CKAN config file, you'll see something like this in your browser:

Error 500

Server Error

An internal server error occurred

If you have `debug = true` in your CKAN config file, then you'll see a traceback page with details about the crash.

You'll also get a 500 `Server Error` if you try to create a group using the `group_create` API action.

To handle the situation where the site has no `curators` group without crashing, we'll have to handle the exception that CKAN's `member_list()` function raises when it's asked to list the members of a group that doesn't exist. Replace the `member_list` line in your `plugin.py` file with these lines:

```
try:
    members = toolkit.get_action('member_list')(
        data_dict={'id': 'curators', 'object_type': 'user'})
except toolkit.ObjectNotFound:
    # The curators group doesn't exist.
    return {'success': False,
            'msg': "The curators groups doesn't exist, so only sysadmins "
                   "are authorized to create groups."}
```

With these `try` and `except` clauses added, we should be able to load the `/group` page and add groups, even if there isn't already a group called `curators`.

Second, `plugin.py` will crash if a user who is not logged-in tries to create a group. If you logout of CKAN, and then visit `/group/new` you'll see another 500 `Server Error`. You'll also get this error if you post to the `group_create()` API action without *providing an API key*.

When the user isn't logged in, `context['user']` contains the user's IP address instead of a user name:

```
{'model': <module 'ckan.model' from ...>,
 'user': u'127.0.0.1'}
```

When we pass this IP address as the user name to `convert_user_name_or_id_to_id()`, the converter function will raise an exception because no user with that user name exists. We need to handle that exception as well, replace the `convert_user_name_or_id_to_id` line in your `plugin.py` file with these lines:

```
convert_user_name_or_id_to_id = toolkit.get_converter(
    'convert_user_name_or_id_to_id')
try:
    user_id = convert_user_name_or_id_to_id(user_name, context)
except toolkit.Invalid:
    # The user doesn't exist (e.g. they're not logged-in).
    return {'success': False,
            'msg': 'You must be logged-in as a member of the curators '
                   'group to create new groups.'}
```

We're done!

Here's our final, working `plugin.py` module in full:

```
import ckan.plugins as plugins
import ckan.plugins.toolkit as toolkit

def group_create(context, data_dict=None):
    # Get the user name of the logged-in user.
```

```

user_name = context['user']

# Get a list of the members of the 'curators' group.
try:
    members = toolkit.get_action('member_list')(
        data_dict={'id': 'curators', 'object_type': 'user'})
except toolkit.ObjectNotFound:
    # The curators group doesn't exist.
    return {'success': False,
            'msg': "The curators groups doesn't exist, so only sysadmins "
                   "are authorized to create groups."}

# 'members' is a list of (user_id, object_type, capacity) tuples, we're
# only interested in the user_ids.
member_ids = [member_tuple[0] for member_tuple in members]

# We have the logged-in user's user name, get their user id.
convert_user_name_or_id_to_id = toolkit.get_converter(
    'convert_user_name_or_id_to_id')
try:
    user_id = convert_user_name_or_id_to_id(user_name, context)
except toolkit.Invalid:
    # The user doesn't exist (e.g. they're not logged-in).
    return {'success': False,
            'msg': 'You must be logged-in as a member of the curators '
                   'group to create new groups.'}

# Finally, we can test whether the user is a member of the curators group.
if user_id in member_ids:
    return {'success': True}
else:
    return {'success': False,
            'msg': 'Only curators are allowed to create groups'}

class ExampleIAuthFunctionsPlugin(plugins.SingletonPlugin):
    plugins.implements(plugins.IAuthFunctions)

    def get_auth_functions(self):
        return {'group_create': group_create}

```

In working through this tutorial, you've covered all the key concepts needed for writing CKAN extensions, including:

- Creating an extension
- Creating a plugin within your extension
- Adding your plugin to your extension's `setup.py` file, and installing your extension
- Making your plugin implement one of CKAN's *plugin interfaces*
- Using the *plugins toolkit*
- Handling exceptions

Troubleshooting

AttributeError

If you get an `AttributeError` like this one:

```
AttributeError: 'ExampleIAuthFunctionsPlugin' object has no attribute 'get_auth_functions'
```

it means that your plugin class does not implement one of the plugin interface's methods. A plugin must implement every method of every plugin interface that it implements.

Todo

Can you use `inherit=True` to avoid having to implement them all?

Other `AttributeErrors` can happen if your method returns the wrong type of value, check the documentation for each plugin interface method to see what your method should return.

TypeError

If you get a `TypeError` like this one:

```
TypeError: get_auth_functions() takes exactly 3 arguments (1 given)
```

it means that one of your plugin methods has the wrong number of parameters. A plugin has to implement each method in a plugin interface with the same parameters as in the interface.

Using custom config settings in extensions

Extensions can define their own custom config settings that users can add to their CKAN config files to configure the behavior of the extension.

Continuing with the `IAuthFunctions` example from *Writing extensions tutorial*, let's make an alternative version of the extension that allows users to create new groups if a new config setting `ckan.iauthfunctions.users_can_create_groups` is `True`:

```
import pylons.config as config

import ckan.plugins as plugins
import ckan.plugins.toolkit as toolkit

def group_create(context, data_dict=None):

    # Get the value of the ckan.iauthfunctions.users_can_create_groups
    # setting from the CKAN config file as a string, or False if the setting
    # isn't in the config file.
    users_can_create_groups = config.get(
        'ckan.iauthfunctions.users_can_create_groups', False)

    # Convert the value from a string to a boolean.
    users_can_create_groups = toolkit.asbool(users_can_create_groups)

    if users_can_create_groups:
        return {'success': True}
    else:
        return {'success': False,
            'msg': 'Only sysadmins can create groups'}
```

```
class ExampleIAuthFunctionsPlugin(plugins.SingletonPlugin):
    plugins.implements(plugins.IAuthFunctions)

    def get_auth_functions(self):
        return {'group_create': group_create}
```

The `group_create` authorization function in this plugin uses `pylons.config` to read the setting from the config file, then calls `ckan.plugins.toolkit.asbool()` to convert the value from a string (all config settings values are strings, when read from the file) to a boolean.

Note: There are also `asint()` and `aslist()` functions in the plugins toolkit.

With this plugin enabled, you should find that users can create new groups if you have `ckan.iauthfunctions.users_can_create_groups = True` in the `[app:main]` section of your CKAN config file. Otherwise, only sysadmin users will be allowed to create groups.

Note: Names of config settings provided by extensions should include the name of the extension, to avoid conflicting with core config settings or with config settings from other extensions. See *Names of config settings should include the name of the extension*.

Note: The users still need to be logged-in to create groups. In general creating, updating or deleting content in CKAN requires the user to be logged-in to a registered user account, no matter what the relevant authorization function says.

Making configuration options runtime-editable

Extensions can allow certain configuration options to be edited during *runtime*, as opposed to having to edit the configuration file and restart the server.

Warning: Only configuration options which are not critical, sensitive or could cause the CKAN instance to break should be made runtime-editable. You should only add config options that you are comfortable they can be edited during runtime, such as ones you've added in your own extension, or have reviewed the use of in core CKAN.

Note: Only sysadmin users are allowed to modify runtime-editable configuration options.

In this tutorial we will show how to make changes to our extension to make two configuration options runtime-editable: `ckan.datasets_per_page` and a custom one named `ckanext.example_iconfigurer.test_conf`. You can see the changes in the `example_iconfigurer` extension that's packaged with CKAN. If you haven't done yet, you should check the *Writing extensions tutorial* first.

This tutorial assumes that we have CKAN running on the paster development server at <http://localhost:5000>, and that we are using the *API key* of a sysadmin user.

First of all, let's call the `config_option_list()` API action to see what configuration options are editable during runtime (the `| python -m json.tool` bit at the end is added to format the output nicely):

```
curl -H "Authorization: XXX" http://localhost:5000/api/action/config_option_list | python -m json.tool
{
  "help": "http://localhost:5000/api/3/action/help_show?name=config_option_list",
  "result": [
    "ckan.site_custom_css",
```

```
        "ckan.main_css",
        "ckan.site_title",
        "ckan.site_about",
        "ckan.site_url",
        "ckan.site_logo",
        "ckan.site_description",
        "ckan.site_intro_text",
        "ckan.homepage_style",
        "ckan.hola"
    ],
    "success": true
}
```

We can see that the two options that we want to make runtime-editable are not on the list. Trying to update one of them with the `config_option_update()` action would return an error.

To include them, we need to add them to the schema that CKAN will use to decide which configuration options can be edited safely at runtime. This is done with the `update_config_schema()` method of the `IConfigurer` interface.

Let's have a look at how our extension should look like:

```
import ckan.plugins as plugins
import ckan.plugins.toolkit as toolkit

class ExampleIConfigurerPlugin(plugins.SingletonPlugin):

    plugins.implements(plugins.IConfigurer)

    # IConfigurer

    def update_config_schema(self, schema):

        ignore_missing = toolkit.get_validator('ignore_missing')
        is_positive_integer = toolkit.get_validator('is_positive_integer')

        schema.update({
            # This is an existing CKAN core configuration option, we are just
            # making it available to be editable at runtime
            'ckan.datasets_per_page': [ignore_missing, is_positive_integer],

            # This is a custom configuration option
            'ckanext.example_illustrator.test_conf': [ignore_missing, unicode],
        })

        return schema
```

The `update_config_schema` method will receive the default schema for runtime-editable configuration options used by CKAN core. We can then add keys to it to make new options runtime-editable (or remove them if we don't want them to be runtime-editable). The schema is a dictionary mapping configuration option keys to lists of validator and converter functions to be applied to those keys. To get validator functions defined in CKAN core we use the `get_validator()` function.

Note: Make sure that the first validator applied to each key is the `ignore_missing` one, otherwise this key will need to be always set when updating the configuration.

Restart the web server and do another request to the `config_option_list()` API action:

```
curl -H "Authorization: XXX" http://localhost:5000/api/action/config_option_list | python -m json.tool
{
  "help": "http://localhost:5000/api/3/action/help_show?name=config_option_list",
  "result": [
    "ckan.datasets_per_page",
    "ckanext.example_configurer.test_conf",
    "ckan.site_custom_css",
    "ckan.main_css",
    "ckan.site_title",
    "ckan.site_about",
    "ckan.site_url",
    "ckan.site_logo",
    "ckan.site_description",
    "ckan.site_intro_text",
    "ckan.homepage_style",
    "ckan.hola"
  ],
  "success": true
}
```

Our two new configuration options are available to be edited at runtime. We can test it calling the `config_option_update()` action:

```
curl -X POST -H "Authorization: XXX" http://localhost:5000/api/action/config_option_update -d '{"ckan.datasets_per_page": 5}'
{
  "help": "http://localhost:5001/api/3/action/help_show?name=config_option_update",
  "result": {
    "ckan.datasets_per_page": 5
  },
  "success": true
}
```

The configuration has now been updated. If you visit the main search page at <http://localhost:5000/dataset> only 5 datasets should appear in the results as opposed to the usual 20.

At this point both our configuration options can be updated via the API, but we also want to make them available on the *administration interface* so non-technical users don't need to use the API to change them.

To do so, we will extend the CKAN core template as described in the *Customizing CKAN's templates* documentation.

First add the `update_config()` method to your plugin and register the extension templates folder:

```
import ckan.plugins as plugins
import ckan.plugins.toolkit as toolkit

class ExampleIConfigurerPlugin(plugins.SingletonPlugin):
    plugins.implements(plugins.IConfigurer)

    # IConfigurer

    def update_config(self, config):
        # Add extension templates directory
        toolkit.add_template_directory(config, 'templates')

    def update_config_schema(self, schema):
        ignore_missing = toolkit.get_validator('ignore_missing')
        is_positive_integer = toolkit.get_validator('is_positive_integer')
```

```
schema.update({
    # This is an existing CKAN core configuration option, we are just
    # making it available to be editable at runtime
    'ckan.datasets_per_page': [ignore_missing, is_positive_integer],

    # This is a custom configuration option
    'ckanext.example_configurer.test_conf': [ignore_missing, unicode],
})

return schema
```

Now create a new file `config.html` file under `ckanext/yourextension/templates/admin/` with the following contents:

```
{% ckan_extends %}

{% import 'macros/form.html' as form %}

{% block admin_form %}

    {{ super() }}

    <h3>Custom configuration options </h3>

    {{ form.input('ckan.datasets_per_page', id='field-ckan.datasets_per_page', label=_('Datasets per page')) }}

    {{ form.input('ckanext.example_configurer.test_conf', id='field-ckanext.example_configurer.test_conf', label=_('Test conf')) }}

{% endblock %}

{% block admin_form_help %}

    {{ super() }}

    <p><strong>Datasets per page:</strong> Number of datasets displayed in dataset listings (eg search results)</p>

    <p><strong>Test conf:</strong> An example configuration option, set from an extension.</p>

{% endblock %}
```

This template is extending the default core one. The first block adds two new fields for our configuration options below the existing ones. The second adds a helper text for them on the left hand column.

Restart the server and navigate to <http://localhost:5000/ckan-admin/config>. You should see the new fields at the bottom of the form:

The screenshot shows a web form for CKAN configuration. It includes a 'Custom CSS' section with a text area containing the placeholder 'Customisable css inserted into the page header'. Below this is a 'Homepage' dropdown menu currently set to 'Search, stats, introductory are:'. A section titled 'Custom configuration options' contains two input fields: 'Datasets per page' with the value '5', and 'Test conf' which is empty. At the bottom of the form are two buttons: a red 'Reset' button and a blue 'Update Config' button.

Updating the values on the form should update the configuration as before.

Testing extensions

CKAN extensions can have their own tests that are run using `nosetests` in much the same way as running CKAN's own tests (see [Testing CKAN](#)).

Continuing with our [example `iauthfunctions` extension](#), first we need a CKAN config file to be used when running our tests. Create the file `ckanext-iauthfunctions/test.ini` with the following contents:

```
[app:main]
use = config:../ckan/test-core.ini
```

The `use` line declares that this config file inherits the settings from the config file used to run CKAN's own tests (`../ckan` should be the path to your CKAN source directory, relative to your `test.ini` file).

The `test.ini` file is a CKAN config file just like your `/etc/ckan/default/development.ini` and `/etc/ckan/default/production.ini` files, and it can contain any [CKAN config file settings](#) that you want CKAN to use when running your tests, for example:

```
[app:main]
use = config:../ckan/test-core.ini
ckan.site_title = My Test CKAN Site
ckan.site_description = A test site for testing my CKAN extension
```

Next, make the directory that will contain our test modules:

```
mkdir ckanext-iauthfunctions/ckanext/iauthfunctions/tests/
```

Finally, create the file `ckanext-iauthfunctions/ckanext/iauthfunctions/tests/test_iauthfunctions.py` with the following contents:

```
'''Tests for the ckanext.example_iauthfunctions extension.

'''
import paste.fixture
import pylons.test
import pylons.config as config
import webtest

import ckan.model as model
import ckan.tests.legacy as tests
import ckan.plugins
import ckan.tests.factories as factories

class TestExampleIAuthFunctionsCustomConfigSetting(object):
    '''Tests for the plugin_v5_custom_config_setting module.

    '''
    def _get_app(self, users_can_create_groups):

        # Set the custom config option in pylons.config.
        config['ckan.iauthfunctions.users_can_create_groups'] = (
            users_can_create_groups)

        # Return a test app with the custom config.
        app = ckan.config.middleware.make_app(config['global_conf'], **config)
        app = webtest.TestApp(app)

        ckan.plugins.load('example_iauthfunctions_v5_custom_config_setting')

        return app

    def teardown(self):

        # Remove the custom config option from pylons.config.
        del config['ckan.iauthfunctions.users_can_create_groups']

        # Delete any stuff that's been created in the db, so it doesn't
        # interfere with the next test.
        model.repo.rebuild_db()

    @classmethod
    def teardown_class(cls):
        ckan.plugins.unload('example_iauthfunctions_v5_custom_config_setting')

    def test_sysadmin_can_create_group_when_config_is_False(self):
        app = self._get_app(users_can_create_groups=False)
        sysadmin = factories.Sysadmin()

        tests.call_action_api(app, 'group_create', name='test-group',
                              apikey=sysadmin['apikey'])

    def test_user_cannot_create_group_when_config_is_False(self):
        app = self._get_app(users_can_create_groups=False)
        user = factories.User()

        tests.call_action_api(app, 'group_create', name='test-group',
                              apikey=user['apikey'], status=403)
```

```

def test_visitor_cannot_create_group_when_config_is_False(self):
    app = self._get_app(users_can_create_groups=False)

    tests.call_action_api(app, 'group_create', name='test-group',
                          status=403)

def test_sysadmin_can_create_group_when_config_is_True(self):
    app = self._get_app(users_can_create_groups=True)
    sysadmin = factories.Sysadmin()

    tests.call_action_api(app, 'group_create', name='test-group',
                          apikey=sysadmin['apikey'])

def test_user_can_create_group_when_config_is_True(self):
    app = self._get_app(users_can_create_groups=True)
    user = factories.User()

    tests.call_action_api(app, 'group_create', name='test-group',
                          apikey=user['apikey'])

def test_visitor_cannot_create_group_when_config_is_True(self):
    app = self._get_app(users_can_create_groups=True)

    tests.call_action_api(app, 'group_create', name='test-group',
                          status=403)

class TestExampleIAuthFunctionsPluginV4(object):
    '''Tests for the ckanext.example_iauthfunctions.plugin module.'''

    '''
    @classmethod
    def setup_class(cls):
        '''Nose runs this method once to setup our test class.'''

        # Make the Paste TestApp that we'll use to simulate HTTP requests to
        # CKAN.
        cls.app = paste.fixture.TestApp(pylons.test.pylonsapp)

        # Test code should use CKAN's plugins.load() function to load plugins
        # to be tested.
        ckan.plugins.load('example_iauthfunctions_v4')

    def teardown(self):
        '''Nose runs this method after each test method in our test class.'''

        # Rebuild CKAN's database after each test method, so that each test
        # method runs with a clean slate.
        model.repo.rebuild_db()

    @classmethod
    def teardown_class(cls):
        '''Nose runs this method once after all the test methods in our class
        have been run.'''

        '''
        # We have to unload the plugin we loaded, so it doesn't affect any
        # tests that run after ours.

```

```
kan.plugins.unload('example_iauthfunctions_v4')

def _make_curators_group(self):
    '''This is a helper method for test methods to call when they want
    the 'curators' group to be created.'''

    '''
    sysadmin = factories.Sysadmin()

    # Create a user who will *not* be a member of the curators group.
    noncurator = factories.User()

    # Create a user who will be a member of the curators group.
    curator = factories.User()

    # Create the curators group, with the 'curator' user as a member.
    users = [{'name': curator['name'], 'capacity': 'member'}]
    curators_group = tests.call_action_api(self.app, 'group_create',
                                           apikey=sysadmin['apikey'],
                                           name='curators',
                                           users=users)

    return (noncurator, curator, curators_group)

def test_group_create_with_no_curators_group(self):
    '''Test that group_create doesn't crash when there's no curators group.'''

    '''
    sysadmin = factories.Sysadmin()

    # Make sure there's no curators group.
    assert 'curators' not in tests.call_action_api(self.app, 'group_list')

    # Make our sysadmin user create a group. CKAN should not crash.
    tests.call_action_api(self.app, 'group_create', name='test-group',
                         apikey=sysadmin['apikey'])

def test_group_create_with_visitor(self):
    '''A visitor (not logged in) should not be able to create a group.'''

    Note: this also tests that the group_create auth function doesn't
    crash when the user isn't logged in.

    '''
    noncurator, curator, curators_group = self._make_curators_group()
    result = tests.call_action_api(self.app, 'group_create',
                                   name='this_group_should_not_be_created',
                                   status=403)
    assert result['__type'] == 'Authorization Error'

def test_group_create_with_non_curator(self):
    '''A user who isn't a member of the curators group should not be able
    to create a group.'''

    '''
    noncurator, curator, curators_group = self._make_curators_group()
    result = tests.call_action_api(self.app, 'group_create',
                                   name='this_group_should_not_be_created',
```

```

        apikey=noncurator['apikey'],
        status=403)
    assert result['__type'] == 'Authorization Error'

def test_group_create_with_curator(self):
    '''A member of the curators group should be able to create a group.

    '''
    noncurator, curator, curators_group = self._make_curators_group()
    name = 'my-new-group'
    result = tests.call_action_api(self.app, 'group_create',
                                   name=name,
                                   apikey=curator['apikey'])

    assert result['name'] == name

```

To run these extension tests, `cd` into the `ckanext-iauthfunctions` directory and run this command:

```
nosetests --ckan --with-pylons=test.ini ckanext/iauthfunctions/tests
```

Some notes on how these tests work:

- Nose has lots of useful functions for testing, see the [nose documentation](#).
- We're using a `paste.fixture.TestApp` object to simulate sending HTTP requests to the CKAN API or frontend. See [Testing Applications with Paste](#) for some documentation of this.
- We're calling `ckan.tests.call_action_api()` to post (simulated) HTTP requests to the CKAN API. This is a convenience function that CKAN provides for its own tests.
- You might also find it useful to read the [Pylons testing documentation](#).
- The Pylons book also has a [chapter on testing](#).
- Avoid importing the plugin modules directly into your test modules (e.g from `example_iauthfunctions` import `plugin_v5_custom_config_setting`). This causes the plugin to be registered and loaded before the entire test run, so the plugin will be loaded for all tests. This can cause conflicts and test failures.

Todo

Link to CKAN guidelines for *how* to write tests, once those guidelines have been written. Also add any more extension-specific testing details here.

Best practices for writing extensions

Follow CKAN's coding standards

See *Contributing guide*.

Use the plugins toolkit instead of importing CKAN

Try to limit your extension to interacting with CKAN only through CKAN's *plugin interfaces* and *plugins toolkit*. It's a good idea to keep your extension code separate from CKAN as much as possible, so that internal changes in CKAN from one release to the next don't break your extension.

Don't edit CKAN's database tables

An extension can create its own tables in the CKAN database, but it should *not* write to core CKAN tables directly, add columns to core tables, or use foreign keys against core tables.

Implement each plugin class in a separate Python module

This keeps CKAN's plugin loading order simple, see [ckan.plugins](#).

Names of config settings should include the name of the extension

Names of config settings provided by extensions should include the name of the extension, to avoid conflicting with core config settings or with config settings from other extensions. For example:

```
ckan.my_extension.show_most_popular_groups = True
```

Internationalize user-visible strings

All user-visible strings should be internationalized, see [String internationalization](#).

Customizing dataset and resource metadata fields using IDatasetForm

Storing additional metadata for a dataset beyond the default metadata in CKAN is a common use case. CKAN provides a simple way to do this by allowing you to store arbitrary key/value pairs against a dataset when creating or updating the dataset. These appear under the “Additional Information” section on the web interface and in ‘extras’ field of the JSON when accessed via the API.

Default extras can only take strings for their keys and values, no validation is applied to the inputs and you cannot make them mandatory or restrict the possible values to a defined list. By using CKAN's IDatasetForm plugin interface, a CKAN plugin can add custom, first-class metadata fields to CKAN datasets, and can do custom validation of these fields.

See also:

In this tutorial we are assuming that you have read the [Writing extensions tutorial](#)

CKAN schemas and validation

When a dataset is created, updated or viewed, the parameters passed to CKAN (e.g. via the web form when creating or updating a dataset, or posted to an API end point) are validated against a schema. For each parameter, the schema will contain a corresponding list of functions that will be run against the value of the parameter. Generally these functions are used to validate the value (and raise an error if the value fails validation) or convert the value to a different value.

For example, the schemas can allow optional values by using the `ignore_missing()` validator or check that a dataset exists using `package_id_exists()`. A list of available validators can be found at the [Validator functions reference](#). You can also define your own [Custom validators](#).

We will be customizing these schemas to add our additional fields. The `IDatasetForm` interface allows us to override the schemas for creation, updating and displaying of datasets.

<code>create_package_schema()</code>	Return the schema for validating new dataset dicts.
<code>update_package_schema()</code>	Return the schema for validating updated dataset dicts.
<code>show_package_schema()</code>	Return a schema to validate datasets before they're shown to the user.
<code>is_fallback()</code>	Return True if this plugin is the fallback plugin.
<code>package_types()</code>	Return an iterable of package types that this plugin handles.

CKAN allows you to have multiple `IDatasetForm` plugins, each handling different dataset types. So you could customize the CKAN web front end, for different types of datasets. In this tutorial we will be defining our plugin as the fallback plugin. This plugin is used if no other `IDatasetForm` plugin is found that handles that dataset type.

The `IDatasetForm` also has other additional functions that allow you to provide a custom template to be rendered for the CKAN frontend, but we will not be using them for this tutorial.

Adding custom fields to datasets

Create a new plugin named `ckanext-extrafields` and create a class named `ExampleIDatasetFormPlugins` inside `ckanext-extrafields/ckanext/extrafields/plugins.py` that implements the `IDatasetForm` interface and inherits from `SingletonPlugin` and `DefaultDatasetForm`.

```
import ckan.plugins as p
import ckan.plugins.toolkit as tk
```

```
class ExampleIDatasetFormPlugin(p.SingletonPlugin, tk.DefaultDatasetForm):
    p.implements(p.IDatasetForm)
```

Updating the CKAN schema

The `create_package_schema()` function is used whenever a new dataset is created, we'll want update the default schema and insert our custom field here. We will fetch the default schema defined in `default_create_package_schema()` by running `create_package_schema()`'s super function and update it.

```
def create_package_schema(self):
    # let's grab the default schema in our plugin
    schema = super(ExampleIDatasetFormPlugin, self).create_package_schema()
    #our custom field
    schema.update({
        'custom_text': [tk.get_validator('ignore_missing'),
                        tk.get_converter('convert_to_extras')]
    })
    return schema
```

The CKAN schema is a dictionary where the key is the name of the field and the value is a list of validators and converters. Here we have a validator to tell CKAN to not raise a validation error if the value is missing and a converter to convert the value to and save as an extra. We will want to change the `update_package_schema()` function with the same update code.

```
def update_package_schema(self):
    schema = super(ExampleIDatasetFormPlugin, self).update_package_schema()
    #our custom field
    schema.update({
        'custom_text': [tk.get_validator('ignore_missing'),
```

```
        tk.get_converter('convert_to_extras'])
    })
    return schema
```

The `show_package_schema()` is used when the `package_show()` action is called, we want the `default_show_package_schema` to be updated to include our custom field. This time, instead of converting to an extras field, we want our field to be converted *from* an extras field. So we want to use the `convert_from_extras()` converter.

```
def show_package_schema(self):
    schema = super(ExampleIDatasetFormPlugin, self).show_package_schema()
    schema.update({
        'custom_text': [tk.get_converter('convert_from_extras'),
                        tk.get_validator('ignore_missing')]
    })
    return schema
```

Dataset types

The `package_types()` function defines a list of dataset types that this plugin handles. Each dataset has a field containing its type. Plugins can register to handle specific types of dataset and ignore others. Since our plugin is not for any specific type of dataset and we want our plugin to be the default handler, we update the plugin code to contain the following:

```
def is_fallback(self):
    # Return True to register this plugin as the default handler for
    # package types not handled by any other IDatasetForm plugin.
    return True

def package_types(self):
    # This plugin doesn't handle any special package types, it just
    # registers itself as the default (above).
    return []
```

Updating templates

In order for our new field to be visible on the CKAN front-end, we need to update the templates. Add an additional line to make the plugin implement the `IConfigurer` interface

```
class ExampleIDatasetFormPlugin(p.SingletonPlugin, tk.DefaultDatasetForm):
    p.implements(p.IDatasetForm)
    p.implements(p.IConfigurer)
```

This interface allows to implement a function `update_config()` that allows us to update the CKAN config, in our case we want to add an additional location for CKAN to look for templates. Add the following code to your plugin.

```
def update_config(self, config):
    # Add this plugin's templates dir to CKAN's extra_template_paths, so
    # that CKAN will use this plugin's custom templates.
    tk.add_template_directory(config, 'templates')
```

You will also need to add a directory under your extension directory to store the templates. Create a directory called `ckanext-extrafields/ckanext-extrafields/templates/` and the subdirectories `ckanext-extrafields/ckanext-extrafields/templates/package/snippets/`.

We need to override a few templates in order to get our custom field rendered. A common option when using a custom schema is to remove the default custom field handling that allows arbitrary key/value pairs. Create a template file in our templates directory called `package/snippets/package_metadata_fields.html` containing

```
{% ckan_extends %}

{# You could remove 'free extras' from the package form like this, but we keep them for this example
{% block custom_fields %}
    {% endblock %}
#}
```

This overrides the `custom_fields` block with an empty block so the default CKAN custom fields form does not render.

New in version 2.3: Starting from CKAN 2.3 you can combine free extras with custom fields handled with `convert_to_extras` and `convert_from_extras`. On prior versions you'll always need to remove the free extras handling.

Next add a template in our template directory called `package/snippets/package_basic_fields.html` containing

```
{% ckan_extends %}

{% block package_basic_fields_custom %}
    {{ form.input('custom_text', label=_('Custom Text'), id='field-custom_text', placeholder=_('custom
{% endblock %}
```

This adds our `custom_text` field to the editing form. Finally we want to display our `custom_text` field on the dataset page. Add another file called `package/snippets/additional_info.html` containing

```
{% ckan_extends %}

{% block extras %}
    {% if pkg_dict.custom_text %}
        <tr>
            <th scope="row" class="dataset-label">{{ _("Custom Text") }}</th>
            <td class="dataset-details">{{ pkg_dict.custom_text }}</td>
        </tr>
    {% endif %}
{% endblock %}
```

This template overrides the default extras rendering on the dataset page and replaces it to just display our custom field.

You're done! Make sure you have your plugin installed and setup as in the *extension/tutorial*. Then run a development server and you should now have an additional field called "Custom Text" when displaying and adding/editing a dataset.

Cleaning up the code

Before we continue further, we can clean up the `create_package_schema()` and `update_package_schema()`. There is a bit of duplication that we could remove. Replace the two functions with:

```
def _modify_package_schema(self, schema):
    schema.update({
        'custom_text': [tk.get_validator('ignore_missing'),
                        tk.get_converter('convert_to_extras')]
    })
    return schema

def create_package_schema(self):
```

```
schema = super(ExampleIDatasetFormPlugin, self).create_package_schema()
schema = self._modify_package_schema(schema)
return schema

def update_package_schema(self):
    schema = super(ExampleIDatasetFormPlugin, self).update_package_schema()
    schema = self._modify_package_schema(schema)
    return schema
```

Custom validators

You may define custom validators in your extensions and you can share validators between extensions by registering them with the `IValidators` interface.

Any of the following objects may be used as validators as part of a custom dataset, group or organization schema. CKAN's validation code will check for and attempt to use them in this order:

1. a `formencode Validator class` (not discussed)
2. a `formencode Validator instance` (not discussed)
3. a callable object taking a single parameter: `validator(value)`
4. a callable object taking four parameters: `validator(key, flattened_data, errors, context)`
5. a callable object taking two parameters `validator(value, context)`

`validator(value)`

The simplest form of validator is a callable taking a single parameter. For example:

```
from ckan.plugins.toolkit import Invalid

def starts_with_b(value):
    if not value.startswith('b'):
        raise Invalid("Doesn't start with b")
    return value
```

The `starts_with_b` validator causes a validation error for values not starting with 'b'. On a web form this validation error would appear next to the field to which the validator was applied.

`return value` must be used by validators when accepting data or the value will be converted to `None`. This form is useful for converting data as well, because the return value will replace the field value passed:

```
def embiggen(value):
    return value.upper()
```

The `embiggen` validator will convert values passed to all-uppercase.

`validator(value, context)`

Validators that need access to the database or information about the user may be written as a callable taking two parameters. `context['session']` is the sqlalchemy session object and `context['user']` is the username of the logged-in user:

```
from ckan.plugins.toolkit import Invalid

def fred_only(value, context):
    if value and context['user'] != 'fred':
        raise Invalid('only fred may set this value')
    return value
```

Otherwise this is the same as the single-parameter form above.

validator(key, flattened_data, errors, context)

Validators that need to access or update multiple fields may be written as a callable taking four parameters.

This form of validator is passed the all the fields and errors in a “flattened” form. Validator must fetch values from flattened_data may replace values in flattened_data. The return value from this function is ignored.

key is the flattened key for the field to which this validator was applied. For example ('notes',) for the dataset notes field or ('resources', 0, 'url') for the url of the first resource of the dataset. These flattened keys are the same in both the flattened_data and errors dicts passed.

errors contains lists of validation errors for each field.

context is the same value passed to the two-parameter form above.

Note that this form can be tricky to use because some of the values in flattened_data will have had validators applied but other fields won't. You may add this type of validator to the special schema fields '__before' or '__after' to have them run before or after all the other validation takes place to avoid the problem of working with partially-validated data.

Tag vocabularies

If you need to add a custom field where the input options are restricted to a provided list of options, you can use tag vocabularies *Tag Vocabularies*. We will need to create our vocabulary first. By calling vocabulary_create(). Add a function to your plugin.py above your plugin class.

```
def create_country_codes():
    user = tk.get_action('get_site_user')({'ignore_auth': True}, {})
    context = {'user': user['name']}
    try:
        data = {'id': 'country_codes'}
        tk.get_action('vocabulary_show')(context, data)
    except tk.ObjectNotFound:
        data = {'name': 'country_codes'}
        vocab = tk.get_action('vocabulary_create')(context, data)
        for tag in (u'uk', u'ie', u'de', u'fr', u'es'):
            data = {'name': tag, 'vocabulary_id': vocab['id']}
            tk.get_action('tag_create')(context, data)
```

This code block is taken from the example_idatsetform plugin. create_country_codes tries to fetch the vocabulary country_codes using vocabulary_show(). If it is not found it will create it and iterate over the list of countries 'uk', 'ie', 'de', 'fr', 'es'. For each of these a vocabulary tag is created using tag_create(), belonging to the vocabulary country_code.

Although we have only defined five tags here, additional tags can be created at any point by a sysadmin user by calling tag_create() using the API or action functions. Add a second function below create_country_codes

```
def country_codes():
    create_country_codes()
    try:
        tag_list = tk.get_action('tag_list')
        country_codes = tag_list(data_dict={'vocabulary_id': 'country_codes'})
        return country_codes
    except tk.ObjectNotFound:
        return None
```

country_codes will call create_country_codes so that the country_codes vocabulary is created if it does not exist. Then it calls tag_list() to return all of our vocabulary tags together. Now we have a way of retrieving our tag vocabularies and creating them if they do not exist. We just need our plugin to call this code.

Adding tags to the schema

Update _modify_package_schema() and show_package_schema()

```
def _modify_package_schema(self, schema):
    schema.update({
        'custom_text': [tk.get_validator('ignore_missing'),
                        tk.get_converter('convert_to_extras')]
    })
    schema.update({
        'country_code': [
            tk.get_validator('ignore_missing'),
            tk.get_converter('convert_to_tags')('country_codes')
        ]
    })
    return schema

def show_package_schema(self):
    schema = super(ExampleIDatasetFormPlugin, self).show_package_schema()
    schema.update({
        'custom_text': [tk.get_converter('convert_from_extras'),
                        tk.get_validator('ignore_missing')]
    })

    schema['tags']['__extras'].append(tk.get_converter('free_tags_only'))
    schema.update({
        'country_code': [
            tk.get_converter('convert_from_tags')('country_codes'),
            tk.get_validator('ignore_missing')]
    })
    return schema
```

We are adding our tag to our plugin's schema. A converter is required to convert the field in to our tag in a similar way to how we converted our field to extras earlier. In show_package_schema() we convert from the tag back again but we have an additional line with another converter containing free_tags_only(). We include this line so that vocab tags are not shown mixed with normal free tags.

Adding tags to templates

Add an additional plugin.implements line to to your plugin to implement the ITemplateHelpers, we will need to add a get_helpers() function defined for this interface.

```
p.implements(p.ITemplateHelpers)
```

```
def get_helpers(self):
    return {'country_codes': country_codes}
```

Our intention here is to tie our country_code fetching/creation to when they are used in the templates. Add the code below to package/snippets/package_metadata_fields.html

```
#}
```

```
{% block package_metadata_fields %}
```

```
<div class="control-group">
  <label class="control-label" for="field-country_code">{{ _("Country Code") }}</label>
  <div class="controls">
    <select id="field-country_code" name="country_code" data-module="autocomplete">
      {% for country_code in h.country_codes() %}
        <option value="{{ country_code }}" {% if country_code in data.get('country_code', []) %}selected{% endif %}>{{ country_code }}</option>
      {% endfor %}
    </select>
  </div>
</div>
```

```
{{ super() }}
```

```
{% endblock %}
```

This adds our country code to our template, here we are using the additional helper country_codes that we defined in our get_helpers function in our plugin.

Adding custom fields to resources

In order to customize the fields in a resource the schema for resources needs to be modified in a similar way to the datasets. The resource schema is nested in the dataset dict as package['resources']. We modify this dict in a similar way to the dataset schema. Change _modify_package_schema to the following.

```
def _modify_package_schema(self, schema):
    # Add our custom country_code metadata field to the schema.
    schema.update({
        'country_code': [tk.get_validator('ignore_missing'),
                        tk.get_converter('convert_to_tags')('country_codes')]
    })
    # Add our custom_test metadata field to the schema, this one will use
    # convert_to_extras instead of convert_to_tags.
    schema.update({
        'custom_text': [tk.get_validator('ignore_missing'),
                       tk.get_converter('convert_to_extras')]
    })
    # Add our custom_resource_text metadata field to the schema
    schema['resources'].update({
        'custom_resource_text': [tk.get_validator('ignore_missing')]
    })
    return schema
```

Update show_package_schema() similarly

```
def show_package_schema(self):
    schema = super(ExampleIDatasetFormPlugin, self).show_package_schema()

    # Don't show vocab tags mixed in with normal 'free' tags
    # (e.g. on dataset pages, or on the search page)
    schema['tags']['__extras'].append(tk.get_converter('free_tags_only'))

    # Add our custom country_code metadata field to the schema.
    schema.update({
        'country_code': [
            tk.get_converter('convert_from_tags')('country_codes'),
            tk.get_validator('ignore_missing')]
    })

    # Add our custom_text field to the dataset schema.
    schema.update({
        'custom_text': [tk.get_converter('convert_from_extras'),
            tk.get_validator('ignore_missing')]
    })

    schema['resources'].update({
        'custom_resource_text' : [ tk.get_validator('ignore_missing') ]
    })
    return schema

# These methods just record how many times they're called, for testing
# purposes.
# TODO: It might be better to test that custom templates returned by
# these methods are actually used, not just that the methods get
# called.
```

Save and reload your development server CKAN will take any additional keys from the resource schema and save them the its extras field. The templates will automatically check this field and display them in the resource_read page.

Sorting by custom fields on the dataset search page

Now that we've added our custom field, we can customize the CKAN web front end search page to sort datasets by our custom field. Add a new file called `ckanext-extrafields/ckanext/extrafields/templates/package/search.html` containing:

```
{% ckan_extends %}

{% block form %}
    {% set facets = {
        'fields': c.fields_grouped,
        'search': c.search_facets,
        'titles': c.facet_titles,
        'translated_fields': c.translated_fields,
        'remove_field': c.remove_field }
    %}
    {% set sorting = [
        _('Relevance'), 'score desc, metadata_modified desc'),
        _('Name Ascending'), 'title_string asc'),
        _('Name Descending'), 'title_string desc'),
        _('Last Modified'), 'metadata_modified desc'),
        _('Custom Field Ascending'), 'custom_text asc'),
```

```

        (_('Custom Field Descending'), 'custom_text desc'),
        (_('Popular'), 'views_recent desc') if g.tracking_enabled else (false, false) ]
    %}
    {% snippet 'snippets/search_form.html', type='dataset', query=c.q, sorting=sorting, sorting_select=
{% endblock %}

```

This overrides the search ordering drop down code block, the code is the same as the default dataset search block but we are adding two additional lines that define the display name of that search ordering (e.g. Custom Field Ascending) and the SOLR sort ordering (e.g. custom_text asc). If you reload your development server you should be able to see these two additional sorting options on the dataset search page.

The SOLR sort ordering can define arbitrary functions for custom sorting, but this is beyond the scope of this tutorial for further details see <http://wiki.apache.org/solr/CommonQueryParameters#sort> and <http://wiki.apache.org/solr/FunctionQuery>

You can find the complete source for this tutorial at https://github.com/ckan/ckan/tree/master/ckanext/example_idatasetform

Plugin interfaces reference

`ckan.plugins` contains a few core classes and functions for plugins to use:

`ckan.plugins`

class `ckan.plugins.SingletonPlugin(**kwargs)`

Base class for plugins which are singletons (ie most of them)

One singleton instance of this class will be created when the plugin is loaded. Subsequent calls to the class constructor will always return the same singleton instance.

class `ckan.plugins.Plugin(**kwargs)`

Base class for plugins which require multiple instances.

Unless you need multiple instances of your plugin object you should probably use `SingletonPlugin`.

`ckan.plugins.implements` (*interface, namespace=None, inherit=False, service=True*)

Can be used in the class definition of *Plugin* subclasses to declare the extension points that are implemented by this interface class.

If the *inherits* option is *True*, then this *Plugin* class inherits from the *interface* class.

`ckan.plugins.interfaces`

A collection of interfaces that CKAN plugins can implement to customize and extend CKAN.

class `ckan.plugins.interfaces.IMiddleware`

Hook into Pylons middleware stack

make_middleware (*app, config*)

Return an app configured with this middleware

make_error_log_middleware (*app, config*)

Return an app configured with this error log middleware

class `ckan.plugins.interfaces.IGenshiStreamFilter`

Hook into template rendering. See `ckan.lib.base.py:render`

filter (*stream*)

Return a filtered Genshi stream. Called when any page is rendered.

Parameters *stream* – Genshi stream of the current output document

Returns filtered Genshi stream

class `ckan.plugins.interfaces.IRoutes`

Plugin into the setup of the routes map creation.

before_map (*map*)

Called before the routes map is generated. `before_map` is before any other mappings are created so can override all other mappings.

Parameters *map* – Routes map object

Returns Modified version of the map object

after_map (*map*)

Called after routes map is set up. `after_map` can be used to add fall-back handlers.

Parameters *map* – Routes map object

Returns Modified version of the map object

class `ckan.plugins.interfaces.IMapper`

A subset of the SQLAlchemy mapper extension hooks. See <http://www.sqlalchemy.org/docs/05/reference/orm/interfaces.html#sqla>

Example:

```
>>> class MyPlugin(SingletonPlugin):
...     implements(IMapper)
...
...     def after_update(self, mapper, connection, instance):
...         log("Updated: %r", instance)
```

before_insert (*mapper, connection, instance*)

Receive an object instance before that instance is INSERTed into its table.

before_update (*mapper, connection, instance*)

Receive an object instance before that instance is UPDATEed.

before_delete (*mapper, connection, instance*)

Receive an object instance before that instance is DELETEed.

after_insert (*mapper, connection, instance*)

Receive an object instance after that instance is INSERTed.

after_update (*mapper, connection, instance*)

Receive an object instance after that instance is UPDATEed.

after_delete (*mapper, connection, instance*)

Receive an object instance after that instance is DELETEed.

class `ckan.plugins.interfaces.ISession`

A subset of the SQLAlchemy session extension hooks.

after_begin (*session, transaction, connection*)

Execute after a transaction is begun on a connection

before_flush (*session, flush_context, instances*)

Execute before flush process has started.

after_flush (*session*, *flush_context*)
Execute after flush has completed, but before commit has been called.

before_commit (*session*)
Execute right before commit is called.

after_commit (*session*)
Execute after a commit has occurred.

after_rollback (*session*)
Execute after a rollback has occurred.

class `ckan.plugins.interfaces.IDomainObjectModification`
Receives notification of new, changed and deleted datasets.

class `ckan.plugins.interfaces.IResourceUrlChange`
Receives notification of changed urls.

class `ckan.plugins.interfaces.IResourceView`
Add custom view renderings for different resource types.

info ()
Returns a dictionary with configuration options for the view.

The available keys are:

Parameters

- **name** – name of the view type. This should match the name of the actual plugin (eg `image_view` or `recline_view`).
- **title** – title of the view type, will be displayed on the frontend. This should be translatable (ie wrapped on `toolkit._('Title')`).
- **default_title** – default title that will be used if the view is created automatically (optional, defaults to 'View').
- **default_description** – default description that will be used if the view is created automatically (optional, defaults to '').
- **icon** – icon for the view type. Should be one of the [Font Awesome](#) types without the *icon*-prefix eg. `compass` (optional, defaults to 'picture').
- **always_available** – the view type should be always available when creating new views regardless of the format of the resource (optional, defaults to False).
- **iframed** – the view template should be iframed before rendering. You generally want this option to be True unless the view styles and JavaScript don't clash with the main site theme (optional, defaults to True).
- **preview_enabled** – the preview button should appear on the edit view form. Some view types have their previews integrated with the form (optional, defaults to False).
- **full_page_edit** – the edit form should take the full page width of the page (optional, defaults to False).
- **schema** – schema to validate extra configuration fields for the view (optional). Schemas are defined as a dictionary, with the keys being the field name and the values a list of validator functions that will get applied to the field. For instance:

```
{
    'offset': [ignore_empty, natural_number_validator],
```

```
        'limit': [ignore_empty, natural_number_validator],
    }
```

Example configuration object:

```
{'name': 'image_view',
 'title': toolkit._('Image'),
 'schema': {
     'image_url': [ignore_empty, unicode]
 },
 'icon': 'picture',
 'always_available': True,
 'iframed': False,
}
```

Returns a dictionary with the view type configuration

Return type dict

can_view (*data_dict*)

Returns whether the plugin can render a particular resource.

The *data_dict* contains the following keys:

Parameters

- **resource** – dict of the resource fields
- **package** – dict of the full parent dataset

Returns True if the plugin can render a particular resource, False otherwise

Return type bool

setup_template_variables (*context*, *data_dict*)

Adds variables to be passed to the template being rendered.

This should return a new dict instead of updating the input *data_dict*.

The *data_dict* contains the following keys:

Parameters

- **resource_view** – dict of the resource view being rendered
- **resource** – dict of the parent resource fields
- **package** – dict of the full parent dataset

Returns a dictionary with the extra variables to pass

Return type dict

view_template (*context*, *data_dict*)

Returns a string representing the location of the template to be rendered when the view is displayed

The path will be relative to the template directory you registered using the `add_template_directory()` on the `update_config` method, for instance `views/my_view.html`.

Parameters

- **resource_view** – dict of the resource view being rendered
- **resource** – dict of the parent resource fields

- **package** – dict of the full parent dataset

Returns the location of the view template.

Return type string

form_template (*context*, *data_dict*)

Returns a string representing the location of the template to be rendered when the edit view form is displayed

The path will be relative to the template directory you registered using the `add_template_directory()` on the `update_config` method, for instance `views/my_view_form.html`.

Parameters

- **resource_view** – dict of the resource view being rendered
- **resource** – dict of the parent resource fields
- **package** – dict of the full parent dataset

Returns the location of the edit view form template.

Return type string

class `ckan.plugins.interfaces.IResourcePreview`

Warning: This interface is deprecated, and is only kept for backwards compatibility with the old resource preview code. Please use `IResourceView` for writing custom view plugins.

can_preview (*data_dict*)

Return info on whether the plugin can preview the resource.

This can be done in two ways:

1. The old way is to just return `True` or `False`.

2. The new way is to return a dict with three keys:

- **can_preview** (boolean) `True` if the extension can preview the resource.
- **fixable** (string) A string explaining how preview for the resource could be enabled, for example if the `resource_proxy` plugin was enabled.
- **quality** (int) How good the preview is: 1 (poor), 2 (average) or 3 (good). When multiple preview extensions can preview the same resource, this is used to determine which extension will be used.

Parameters **data_dict** (*dictionary*) – the resource to be previewed and the dataset that it belongs to.

Make sure to check the `on_same_domain` value of the resource or the url if your preview requires the resource to be on the same domain because of the same-origin policy. To find out how to preview resources that are on a different domain, read [Resource Proxy](#).

setup_template_variables (*context*, *data_dict*)

Add variables to `c` just prior to the template being rendered. The `data_dict` contains the resource and the package.

Change the url to a proxied domain if necessary.

preview_template (*context*, *data_dict*)

Returns a string representing the location of the template to be rendered for the read page. The *data_dict* contains the resource and the package.

class `ckan.plugins.interfaces.ITagController`

Hook into the Tag controller. These will usually be called just before committing or returning the respective object, i.e. all validation, synchronization and authorization setup are complete.

before_view (*tag_dict*)

Extensions will receive this before the tag gets displayed. The dictionary passed will be the one that gets sent to the template.

class `ckan.plugins.interfaces.IGroupController`

Hook into the Group controller. These will usually be called just before committing or returning the respective object, i.e. all validation, synchronization and authorization setup are complete.

before_view (*pkg_dict*)

Extensions will receive this before the group gets displayed. The dictionary passed will be the one that gets sent to the template.

class `ckan.plugins.interfaces.IOrganizationController`

Hook into the Organization controller. These will usually be called just before committing or returning the respective object, i.e. all validation, synchronization and authorization setup are complete.

before_view (*pkg_dict*)

Extensions will receive this before the organization gets displayed. The dictionary passed will be the one that gets sent to the template.

class `ckan.plugins.interfaces.IPackageController`

Hook into the package controller. (see `IGroupController`)

after_create (*context*, *pkg_dict*)

Extensions will receive the validated data dict after the package has been created (Note that the create method will return a package domain object, which may not include all fields). Also the newly created package id will be added to the dict.

after_update (*context*, *pkg_dict*)

Extensions will receive the validated data dict after the package has been updated (Note that the edit method will return a package domain object, which may not include all fields).

after_delete (*context*, *pkg_dict*)

Extensions will receive the data dict (typically containing just the package id) after the package has been deleted.

after_show (*context*, *pkg_dict*)

Extensions will receive the validated data dict after the package is ready for display (Note that the read method will return a package domain object, which may not include all fields).

before_search (*search_params*)

Extensions will receive a dictionary with the query parameters, and should return a modified (or not) version of it.

search_params will include an *extras* dictionary with all values from fields starting with *ext_*, so extensions can receive user input from specific fields.

after_search (*search_results*, *search_params*)

Extensions will receive the search results, as well as the search parameters, and should return a modified (or not) object with the same structure:

```
{ 'count': '', 'results': '', 'facets': '' }
```

Note that count and facets may need to be adjusted if the extension changed the results for some reason.

search_params will include an *extras* dictionary with all values from fields starting with *ext_*, so extensions can receive user input from specific fields.

before_index (*pkg_dict*)

Extensions will receive what will be given to the solr for indexing. This is essentially a flattened dict (except for multi-valued fields such as tags) of all the terms sent to the indexer. The extension can modify this by returning an altered version.

before_view (*pkg_dict*)

Extensions will receive this before the dataset gets displayed. The dictionary passed will be the one that gets sent to the template.

class `ckan.plugins.interfaces.IResourceController`

Hook into the resource controller.

before_create (*context, resource*)

Extensions will receive this before a resource is created.

Parameters

- **context** (*dictionary*) – The context object of the current request, this includes for example access to the model and the user.
- **resource** (*dictionary*) – An object representing the resource to be added to the dataset (the one that is about to be created).

after_create (*context, resource*)

Extensions will receive this after a resource is created.

Parameters

- **context** (*dictionary*) – The context object of the current request, this includes for example access to the model and the user.
- **resource** (*dictionary*) – An object representing the latest resource added to the dataset (the one that was just created). A key in the resource dictionary worth mentioning is *url_type* which is set to *upload* when the resource file is uploaded instead of *linked*.

before_update (*context, current, resource*)

Extensions will receive this before a resource is updated.

Parameters

- **context** (*dictionary*) – The context object of the current request, this includes for example access to the model and the user.
- **current** (*dictionary*) – The current resource which is about to be updated
- **resource** (*dictionary*) – An object representing the updated resource which will replace the current one.

after_update (*context, resource*)

Extensions will receive this after a resource is updated.

Parameters

- **context** (*dictionary*) – The context object of the current request, this includes for example access to the model and the user.
- **resource** (*dictionary*) – An object representing the updated resource in the dataset (the one that was just updated). As with *after_create*, a noteworthy key in the resource

dictionary `url_type` which is set to `upload` when the resource file is uploaded instead of `linked`.

before_delete (*context, resource, resources*)

Extensions will receive this before a previously created resource is deleted.

Parameters

- **context** (*dictionary*) – The context object of the current request, this includes for example access to the `model` and the `user`.
- **resource** (*dictionary*) – An object representing the resource that is about to be deleted. This is a dictionary with one key: `id` which holds the `id string` of the resource that should be deleted.
- **resources** (*list*) – The list of resources from which the resource will be deleted (including the resource to be deleted if it existed in the package).

after_delete (*context, resources*)

Extensions will receive this after a previously created resource is deleted.

Parameters

- **context** (*dictionary*) – The context object of the current request, this includes for example access to the `model` and the `user`.
- **resources** – A list of objects representing the remaining resources after a resource has been removed.

before_show (*resource_dict*)

Extensions will receive the validated data dict before the resource is ready for display.

Be aware that this method is not only called for UI display, but also in other methods like when a resource is deleted because showing a package is used to get access to the resources in a package.

class `ckan.plugins.interfaces.IPluginObserver`

Plugin to the plugin loading mechanism

before_load (*plugin*)

Called before a plugin is loaded This method is passed the plugin class.

after_load (*service*)

Called after a plugin has been loaded. This method is passed the instantiated service object.

before_unload (*plugin*)

Called before a plugin is loaded This method is passed the plugin class.

after_unload (*service*)

Called after a plugin has been unloaded. This method is passed the instantiated service object.

class `ckan.plugins.interfaces.IConfigurable`

Pass configuration to plugins and extensions

configure (*config*)

Called by `load_environment`

class `ckan.plugins.interfaces.IConfigurer`

Configure CKAN (pylons) environment via the `pylons.config` object

update_config (*config*)

Called by `load_environment` at earliest point when `config` is available to plugins. The `config` should be updated in place.

Parameters `config` – `pylons.config` object

update_config_schema (*schema*)

Return a schema with the runtime-editable config options

CKAN will use the returned schema to decide which configuration options can be edited during runtime (using `ckan.logic.action.update.config_option_update()`) and to validate them before storing them.

Defaults to `ckan.logic.schema.default_update_configuration_schema()`, which will be passed to all extensions implementing this method, which can add or remove runtime-editable config options to it.

Parameters *schema* (*dictionary*) – a dictionary mapping runtime-editable configuration option keys to lists of validator and converter functions to be applied to those keys

Returns a dictionary mapping runtime-editable configuration option keys to lists of validator and converter functions to be applied to those keys

Return type dictionary

class `ckan.plugins.interfaces.IActions`

Allow adding of actions to the logic layer.

get_actions ()

Should return a dict, the keys being the name of the logic function and the values being the functions themselves.

By decorating a function with the `ckan.logic.side_effect_free` decorator, the associated action will be made available by a GET request (as well as the usual POST request) through the action API.

class `ckan.plugins.interfaces.IValidators`

Add extra validators to be returned by `ckan.plugins.toolkit.get_validator()`.

get_validators ()

Return the validator functions provided by this plugin.

Return a dictionary mapping validator names (strings) to validator functions. For example:

```
{'valid_shoe_size': shoe_size_validator,
 'valid_hair_color': hair_color_validator}
```

These validator functions would then be available when a plugin calls `ckan.plugins.toolkit.get_validator()`.

class `ckan.plugins.interfaces.IAuthFunctions`

Override CKAN's authorization functions, or add new auth functions.

get_auth_functions ()

Return the authorization functions provided by this plugin.

Return a dictionary mapping authorization function names (strings) to functions. For example:

```
{'user_create': my_custom_user_create_function,
 'group_create': my_custom_group_create}
```

When a user tries to carry out an action via the CKAN API or web interface and CKAN or a CKAN plugin calls `check_access('some_action')` as a result, an authorization function named 'some_action' will be searched for in the authorization functions registered by plugins and in CKAN's core authorization functions (found in `ckan/logic/auth/`).

For example when a user tries to create a package, a 'package_create' authorization function is searched for.

If an extension registers an authorization function with the same name as one of CKAN's default authorization functions (as with 'user_create' and 'group_create' above), the extension's function will override the default one.

Each authorization function should take two parameters `context` and `data_dict`, and should return a dictionary {'success': True} to authorize the action or {'success': False} to deny it, for example:

```
def user_create(context, data_dict=None):
    if (some condition):
        return {'success': True}
    else:
        return {'success': False, 'msg': 'Not allowed to register'}
```

The context object will contain a `model` that can be used to query the database, a `user` containing the name of the user doing the request (or their IP if it is an anonymous web request) and an `auth_user_obj` containing the actual `model.User` object (or `None` if it is an anonymous request).

See `ckan/logic/auth/` for more examples.

Note that by default, all auth functions provided by extensions are assumed to require a validated user or API key, otherwise a `ckan.logic.NotAuthorized` exception will be raised. This check will be performed *before* calling the actual auth function. If you want to allow anonymous access to one of your actions, its auth function must be decorated with the `auth_allow_anonymous_access` decorator, available on the plugins toolkit.

For example:

```
import ckan.plugins as p

@p.toolkit.auth_allow_anonymous_access
def my_search_action(context, data_dict):
    # Note that you can still return {'success': False} if for some
    # reason access is denied.

def my_create_action(context, data_dict):
    # Unless there is a logged in user or a valid API key provided
    # NotAuthorized will be raised before reaching this function.
```

class ckan.plugins.interfaces.ITemplateHelpers

Add custom template helper functions.

By implementing this plugin interface plugins can provide their own template helper functions, which custom templates can then access via the `h` variable.

See `ckanext/example_itemplatehelpers` for an example plugin.

get_helpers()

Return a dict mapping names to helper functions.

The keys of the dict should be the names with which the helper functions will be made available to templates, and the values should be the functions themselves. For example, a dict like: {'example_helper': example_helper} allows templates to access the `example_helper` function via `h.example_helper()`.

Function names should start with the name of the extension providing the function, to prevent name clashes between extensions.

class ckan.plugins.interfaces.IDatasetForm

Customize CKAN's dataset (package) schemas and forms.

By implementing this interface plugins can customise CKAN's dataset schema, for example to add new custom fields to datasets.

Multiple `IDatasetForm` plugins can be used at once, each plugin associating itself with different package types using the `package_types()` and `is_fallback()` methods below, and then providing different schemas and templates for different types of dataset. When a package controller action is invoked, the `type` field of the package will determine which `IDatasetForm` plugin (if any) gets delegated to.

When implementing `IDatasetForm`, you can inherit from `ckan.plugins.toolkit.DefaultDatasetForm`, which provides default implementations for each of the methods defined in this interface.

See `ckanext/example_idatasetform` for an example plugin.

`package_types()`

Return an iterable of package types that this plugin handles.

If a request involving a package of one of the returned types is made, then this plugin instance will be delegated to.

There cannot be two `IDatasetForm` plugins that return the same package type, if this happens then CKAN will raise an exception at startup.

Return type iterable of strings

`is_fallback()`

Return `True` if this plugin is the fallback plugin.

When no `IDatasetForm` plugin's `package_types()` match the `type` of the package being processed, the fallback plugin is delegated to instead.

There cannot be more than one `IDatasetForm` plugin whose `is_fallback()` method returns `True`, if this happens CKAN will raise an exception at startup.

If no `IDatasetForm` plugin's `is_fallback()` method returns `True`, CKAN will use `DefaultDatasetForm` as the fallback.

Return type boolean

`create_package_schema()`

Return the schema for validating new dataset dicts.

CKAN will use the returned schema to validate and convert data coming from users (via the dataset form or API) when creating new datasets, before entering that data into the database.

If it inherits from `ckan.plugins.toolkit.DefaultDatasetForm`, a plugin can call `DefaultDatasetForm's create_package_schema()` method to get the default schema and then modify and return it.

CKAN's `convert_to_tags()` or `convert_to_extras()` functions can be used to convert custom fields into dataset tags or extras for storing in the database.

See `ckanext/example_idatasetform` for examples.

Returns a dictionary mapping dataset dict keys to lists of validator and converter functions to be applied to those keys

Return type dictionary

`update_package_schema()`

Return the schema for validating updated dataset dicts.

CKAN will use the returned schema to validate and convert data coming from users (via the dataset form or API) when updating datasets, before entering that data into the database.

If it inherits from `ckan.plugins.toolkit.DefaultDatasetForm`, a plugin can call `DefaultDatasetForm.update_package_schema()` method to get the default schema and then modify and return it.

CKAN's `convert_to_tags()` or `convert_to_extras()` functions can be used to convert custom fields into dataset tags or extras for storing in the database.

See `ckanext/example_idatasetform` for examples.

Returns a dictionary mapping dataset dict keys to lists of validator and converter functions to be applied to those keys

Return type dictionary

`show_package_schema()`

Return a schema to validate datasets before they're shown to the user.

CKAN will use the returned schema to validate and convert data coming from the database before it is returned to the user via the API or passed to a template for rendering.

If it inherits from `ckan.plugins.toolkit.DefaultDatasetForm`, a plugin can call `DefaultDatasetForm.show_package_schema()` method to get the default schema and then modify and return it.

If you have used `convert_to_tags()` or `convert_to_extras()` in your `create_package_schema()` and `update_package_schema()` then you should use `convert_from_tags()` or `convert_from_extras()` in your `show_package_schema()` to convert the tags or extras in the database back into your custom dataset fields.

See `ckanext/example_idatasetform` for examples.

Returns a dictionary mapping dataset dict keys to lists of validator and converter functions to be applied to those keys

Return type dictionary

`setup_template_variables(context, data_dict)`

Add variables to the template context for use in templates.

This function is called before a dataset template is rendered. If you have custom dataset templates that require some additional variables, you can add them to the template context `ckan.plugins.toolkit.c` here and they will be available in your templates. See `ckanext/example_idatasetform` for an example.

`new_template()`

Return the path to the template for the new dataset page.

The path should be relative to the plugin's templates dir, e.g. `'package/new.html'`.

Return type string

`read_template()`

Return the path to the template for the dataset read page.

The path should be relative to the plugin's templates dir, e.g. `'package/read.html'`.

If the user requests the dataset in a format other than HTML (CKAN supports returning datasets in RDF/XML or N3 format by appending `.rdf` or `.n3` to the dataset read URL, see [Linked Data and RDF](#)) then CKAN will try to render a template file with the same path as returned by this function, but a different filename extension, e.g. `'package/read.rdf'`. If your extension doesn't have this RDF version of the template file, the user will get a 404 error.

Return type string

edit_template()

Return the path to the template for the dataset edit page.

The path should be relative to the plugin's templates dir, e.g. 'package/edit.html'.

Return type string

search_template()

Return the path to the template for use in the dataset search page.

This template is used to render each dataset that is listed in the search results on the dataset search page.

The path should be relative to the plugin's templates dir, e.g. 'package/search.html'.

Return type string

history_template()

Return the path to the template for the dataset history page.

The path should be relative to the plugin's templates dir, e.g. 'package/history.html'.

Return type string

resource_template()

Return the path to the template for the resource read page.

The path should be relative to the plugin's templates dir, e.g. 'package/resource_read.html'.

Return type string

package_form()

Return the path to the template for the dataset form.

The path should be relative to the plugin's templates dir, e.g. 'package/form.html'.

Return type string

resource_form()

Return the path to the template for the resource form.

The path should be relative to the plugin's templates dir, e.g. 'package/snippets/resource_form.html'

Return type string

validate(context, data_dict, schema, action)

Customize validation of datasets.

When this method is implemented it is used to perform all validation for these datasets. The default implementation calls and returns the result from `ckan.plugins.toolkit.navl_validate`.

This is an advanced interface. Most changes to validation should be accomplished by customizing the schemas returned from `show_package_schema()`, `create_package_schema()` and `update_package_schema()`. If you need to have a different schema depending on the user or value of any field stored in the dataset, or if you wish to use a different method for validation, then this method may be used.

Parameters

- **context** (*dictionary*) – extra information about the request
- **data_dict** (*dictionary*) – the dataset to be validated
- **schema** (*dictionary*) – a schema, typically from `show_package_schema()`, `create_package_schema()` or `update_package_schema()`

- **action** (*string*) – 'package_show', 'package_create' or 'package_update'

Returns (data_dict, errors) where data_dict is the possibly-modified dataset and errors is a dictionary with keys matching data_dict and lists-of-string-error-messages as values

Return type (dictionary, dictionary)

class `ckan.plugins.interfaces.IGroupForm`

Allows customisation of the group controller as a plugin.

The behaviour of the plugin is determined by 5 method hooks:

- `package_form(self)`
- `form_to_db_schema(self)`
- `db_to_form_schema(self)`
- `check_data_dict(self, data_dict)`
- `setup_template_variables(self, context, data_dict)`

Furthermore, there can be many implementations of this plugin registered at once. With each instance associating itself with 0 or more group type strings. When a group controller action is invoked, the group type determines which of the registered plugins to delegate to. Each implementation must implement two methods which are used to determine the group-type -> plugin mapping:

- `is_fallback(self)`
- `group_types(self)`

Implementations might want to consider mixing in `ckan.lib.plugins.DefaultGroupForm` which provides default behaviours for the 5 method hooks.

`is_fallback()`

Returns true if this provides the fallback behaviour, when no other plugin instance matches a group's type.

There must be exactly one fallback controller defined, any attempt to register more than one will throw an exception at startup. If there's no fallback registered at startup the `ckan.lib.plugins.DefaultGroupForm` used as the fallback.

`group_types()`

Returns an iterable of group type strings.

If a request involving a group of one of those types is made, then this plugin instance will be delegated to.

There must only be one plugin registered to each group type. Any attempts to register more than one plugin instance to a given group type will raise an exception at startup.

`new_template()`

Returns a string representing the location of the template to be rendered for the 'new' page. Uses the `default_group_type` configuration option to determine which plugin to use the template from.

`index_template()`

Returns a string representing the location of the template to be rendered for the index page. Uses the `default_group_type` configuration option to determine which plugin to use the template from.

`read_template()`

Returns a string representing the location of the template to be rendered for the read page

`history_template()`

Returns a string representing the location of the template to be rendered for the history page

edit_template()

Returns a string representing the location of the template to be rendered for the edit page

package_form()

Returns a string representing the location of the template to be rendered. e.g. "group/new_group_form.html".

form_to_db_schema()

Returns the schema for mapping group data from a form to a format suitable for the database.

db_to_form_schema()

Returns the schema for mapping group data from the database into a format suitable for the form (optional)

check_data_dict(data_dict)

Check if the return data is correct.

raise a `DataError` if not.

setup_template_variables(context, data_dict)

Add variables to `c` just prior to the template being rendered.

validate(context, data_dict, schema, action)

Customize validation of groups.

When this method is implemented it is used to perform all validation for these groups. The default implementation calls and returns the result from `ckan.plugins.toolkit.navl_validate`.

This is an advanced interface. Most changes to validation should be accomplished by customizing the schemas returned from `form_to_db_schema()` and `db_to_form_schema()`. If you need to have a different schema depending on the user or value of any field stored in the group, or if you wish to use a different method for validation, then this method may be used.

Parameters

- **context** (*dictionary*) – extra information about the request
- **data_dict** (*dictionary*) – the group to be validated
- **schema** (*dictionary*) – a schema, typically from `form_to_db_schema()`, or `db_to_form_schema()`
- **action** (*string*) – 'group_show', 'group_create', 'group_update', 'organization_show', 'organization_create' or 'organization_update'

Returns (`data_dict`, `errors`) where `data_dict` is the possibly-modified group and `errors` is a dictionary with keys matching `data_dict` and lists-of-string-error-messages as values

Return type (dictionary, dictionary)

class ckan.plugins.interfaces.IFacets

Customize the search facets shown on search pages.

By implementing this interface plugins can customize the search facets that are displayed for filtering search results on the dataset search page, organization pages and group pages.

The `facets_dict` passed to each of the functions below is an `OrderedDict` in which the keys are CKAN's internal names for the facets and the values are the titles that will be shown for the facets in the web interface. The order of the keys in the dict determine the order that facets appear in on the page. For example:

```
{'groups': _('Groups'),
 'tags': _('Tags'),
 'res_format': _('Formats'),
 'license': _('License')}
```

To preserve ordering, make sure to add new facets to the existing dict rather than updating it, ie do this:

```
facets_dict['groups'] = p.toolkit._('Publisher')
facets_dict['secondary_publisher'] = p.toolkit._('Secondary Publisher')
```

rather than this:

```
facets_dict.update({
    'groups': p.toolkit._('Publisher'),
    'secondary_publisher': p.toolkit._('Secondary Publisher'),
})
```

Dataset searches can be faceted on any field in the dataset schema that it makes sense to facet on. This means any dataset field that is in CKAN's Solr search index, basically any field that you see returned by `package_show()`.

If there are multiple `IFacets` plugins active at once, each plugin will be called (in the order that they're listed in the CKAN config file) and they will each be able to modify the facets dict in turn.

dataset_facets (*facets_dict*, *package_type*)

Modify and return the `facets_dict` for the dataset search page.

The `package_type` is the type of package that these facets apply to. Plugins can provide different search facets for different types of package. See [IDatasetForm](#).

Parameters

- **facets_dict** (*OrderedDict*) – the search facets as currently specified
- **package_type** (*string*) – the package type that these facets apply to

Returns the updated `facets_dict`

Return type `OrderedDict`

group_facets (*facets_dict*, *group_type*, *package_type*)

Modify and return the `facets_dict` for a group's page.

The `package_type` is the type of package that these facets apply to. Plugins can provide different search facets for different types of package. See [IDatasetForm](#).

The `group_type` is the type of group that these facets apply to. Plugins can provide different search facets for different types of group. See [IGroupForm](#).

Parameters

- **facets_dict** (*OrderedDict*) – the search facets as currently specified
- **group_type** (*string*) – the group type that these facets apply to
- **package_type** (*string*) – the package type that these facets apply to

Returns the updated `facets_dict`

Return type `OrderedDict`

organization_facets (*facets_dict*, *organization_type*, *package_type*)

Modify and return the `facets_dict` for an organization's page.

The `package_type` is the type of package that these facets apply to. Plugins can provide different search facets for different types of package. See [IDatasetForm](#).

The `organization_type` is the type of organization that these facets apply to. Plugins can provide different search facets for different types of organization. See [IGroupForm](#).

Parameters

- **facets_dict** (*OrderedDict*) – the search facets as currently specified
- **organization_type** (*string*) – the organization type that these facets apply to
- **package_type** (*string*) – the package type that these facets apply to

Returns the updated `facets_dict`

Return type `OrderedDict`

class `ckan.plugins.interfaces.IAuthenticator`
EXPERIMENTAL

Allows custom authentication methods to be integrated into CKAN. Currently it is experimental and the interface may change.

identify ()

called to identify the user.

If the user is identified then it should set `c.user`: The id of the user `c.userobj`: The actual user object (this may be removed as a requirement in a later release so that access to the model is not required)

login ()

called at login.

logout ()

called at logout.

abort (*status_code, detail, headers, comment*)

called on abort. This allows aborts due to authorization issues to be overridden

Plugins toolkit reference

As well as using the variables made available to them by implementing plugin interfaces, plugins will likely want to be able to use parts of the CKAN core library. To allow this, CKAN provides a stable set of classes and functions that plugins can use safe in the knowledge that this interface will remain stable, backward-compatible and with clear deprecation guidelines when new versions of CKAN are released. This interface is available in CKAN's *plugins toolkit*.

class `ckan.plugins.toolkit.BaseController`
Base class for CKAN controller classes to inherit from.

class `ckan.plugins.toolkit.CkanCommand`
Base class for classes that implement CKAN paster commands to inherit.

class `ckan.plugins.toolkit.CkanVersionException`
Exception raised by `requires_ckan_version()` if the required CKAN version is not available.

class `ckan.plugins.toolkit.DefaultDatasetForm`
The default implementation of `IDatasetForm`.

This class serves two purposes:

1. It provides a base class for plugin classes that implement `IDatasetForm` to inherit from, so they can inherit the default behavior and just modify the bits they need to.
2. It is used as the default fallback plugin when no registered `IDatasetForm` plugin handles the given dataset type and no other plugin has registered itself as the fallback plugin.

Note: `DefaultDatasetForm` doesn't call `implements()`, because we don't want it being registered.

class `ckan.plugins.toolkit.DefaultGroupForm`

Provides a default implementation of the pluggable Group controller behaviour.

This class has 2 purposes:

- it provides a base class for `IGroupForm` implementations to use if only a subset of the method hooks need to be customised.
- it provides the fallback behaviour if no plugin is setup to provide the fallback behaviour.

Note - this isn't a plugin implementation. This is deliberate, as we don't want this being registered.

class `ckan.plugins.toolkit.Invalid`

Exception raised by some validator, converter and dictization functions when the given value is invalid.

class `ckan.plugins.toolkit.NotAuthorized`

Exception raised when the user is not authorized to call the action.

For example `package_create()` raises `NotAuthorized` if the user is not authorized to create packages.

class `ckan.plugins.toolkit.ObjectNotFound`

Exception raised by logic functions when a given object is not found.

For example `package_show()` raises `ObjectNotFound` if no package with the given `id` exists.

class `ckan.plugins.toolkit.UnknownValidator`

Exception raised when a requested validator function cannot be found.

class `ckan.plugins.toolkit.ValidationError`

Exception raised by action functions when validating their given `data_dict` fails.

`ckan.plugins.toolkit._(value)`

The Pylons `_()` function.

The Pylons `_()` function is a reference to the `ugettext()` function. Everywhere in your code where you want strings to be internationalized (made available for translation into different languages), wrap them in the `_()` function, eg.:

```
msg = toolkit._("Hello")
```

`ckan.plugins.toolkit.abort(status_code=None, detail='', headers=None, comment=None)`

Abort the current request immediately by returning an HTTP exception.

This is a wrapper for `pylons.controllers.util.abort()` that adds some CKAN custom behavior, including allowing `IAuthenticator` plugins to alter the abort response, and showing flash messages in the web interface.

`ckan.plugins.toolkit.add_ckan_admin_tab(cls, config, route_name, tab_label, config_var='ckan.admin_tabs')`

Update 'ckan.admin_tabs' dict the passed config dict.

`ckan.plugins.toolkit.add_public_directory(cls, config, relative_path)`

Add a path to the `extra_public_paths` config setting.

The path is relative to the file calling this function.

`ckan.plugins.toolkit.add_resource(cls, path, name)`

Add a Fanstatic resource library to CKAN.

Fanstatic libraries are directories containing static resource files (e.g. CSS, JavaScript or image files) that can be accessed from CKAN.

See [Theming guide](#) for more details.

`ckan.plugins.toolkit.add_template_directory(cls, config, relative_path)`

Add a path to the [extra_template_paths](#) config setting.

The path is relative to the file calling this function.

`ckan.plugins.toolkit.asbool(obj)`

Convert a string from the config file into a boolean.

For example: `if toolkit.asbool(config.get('ckan.legacy_templates', False)):`

`ckan.plugins.toolkit.asint(obj)`

Convert a string from the config file into an int.

For example: `bar = toolkit.asint(config.get('ckan.foo.bar', 0))`

`ckan.plugins.toolkit.aslist(obj, sep=None, strip=True)`

Convert a string from the config file into a list.

For example: `bar = toolkit.aslist(config.get('ckan.foo.bar', []))`

`ckan.plugins.toolkit.auth_allow_anonymous_access(action)`

Flag an auth function as not requiring a logged in user

This means that `check_access` won't automatically raise a `NotAuthorized` exception if an authenticated user is not provided in the context. (The auth function can still return `False` if for some reason access is not granted).

`ckan.plugins.toolkit.auth_disallow_anonymous_access(action)`

Flag an auth function as requiring a logged in user

This means that `check_access` will automatically raise a `NotAuthorized` exception if an authenticated user is not provided in the context, without calling the actual auth function.

`ckan.plugins.toolkit.auth_sysadmins_check(action)`

A decorator that prevents sysadmins from being automatically authorized to call an action function.

Normally sysadmins are allowed to call any action function (for example when they're using the [Action API](#) or the web interface), if the user is a sysadmin the action function's authorization function will not even be called.

If an action function is decorated with this decorator, then its authorization function will always be called, even if the user is a sysadmin.

`ckan.plugins.toolkit.c`

The Pylons template context object.

This object is used to pass request-specific information to different parts of the code in a thread-safe way (so that variables from different requests being executed at the same time don't get confused with each other).

Any attributes assigned to `c` are available throughout the template and application code, and are local to the current request.

`ckan.plugins.toolkit.check_access(action, context, data_dict=None)`

Calls the authorization function for the provided action

This is the only function that should be called to determine whether a user (or an anonymous request) is allowed to perform a particular action.

The function accepts a context object, which should contain a 'user' key with the name of the user performing the action, and optionally a dictionary with extra data to be passed to the authorization function.

For example:

```
check_access('package_update', context, data_dict)
```

If not already there, the function will add an `auth_user_obj` key to the context object with the actual `User` object (in case it exists in the database). This check is only performed once per context object.

Raise `NotAuthorized` if the user is not authorized to call the named action function.

If the user *is* authorized to call the action, return `True`.

Parameters

- **action** (*string*) – the name of the action function, eg. `'package_create'`
- **context** (*dict*) –
- **data_dict** (*dict*) –

Raises `NotAuthorized` if the user is not authorized to call the named action

`ckan.plugins.toolkit.check_ckan_version(cls, min_version=None, max_version=None)`

Return `True` if the CKAN version is greater than or equal to `min_version` and less than or equal to `max_version`, return `False` otherwise.

If no `min_version` is given, just check whether the CKAN version is less than or equal to `max_version`.

If no `max_version` is given, just check whether the CKAN version is greater than or equal to `min_version`.

Parameters

- **min_version** (*string*) – the minimum acceptable CKAN version, eg. `'2.1'`
- **max_version** (*string*) – the maximum acceptable CKAN version, eg. `'2.3'`

`ckan.plugins.toolkit.get_action(action)`

Return the named `ckan.logic.action` function.

For example `get_action('package_create')` will normally return the `ckan.logic.action.create.package_create()` function.

For documentation of the available action functions, see [Action API reference](#).

You should always use `get_action()` instead of importing an action function directly, because `IActions` plugins can override action functions, causing `get_action()` to return a plugin-provided function instead of the default one.

Usage:

```
import ckan.plugins.toolkit as toolkit

# Call the package_create action function:
toolkit.get_action('package_create')(context, data_dict)
```

As the context parameter passed to an action function is commonly:

```
context = {'model': ckan.model, 'session': ckan.model.Session,
          'user': pylons.c.user or pylons.c.author}
```

an action function returned by `get_action()` will automatically add these parameters to the context if they are not defined. This is especially useful for plugins as they should not really be importing parts of ckan eg `ckan.model` and as such do not have access to `model` or `model.Session`.

If a context of `None` is passed to the action function then the default context dict will be created.

Parameters **action** (*string*) – name of the action function to return, eg. `'package_create'`

Returns the named action function

Return type callable

`ckan.plugins.toolkit.get_converter(validator)`

Return a validator function by name.

Parameters *validator* (*string*) – the name of the validator function to return, eg. `'package_name_exists'`

Raises `UnknownValidator` if the named validator is not found

Returns the named validator function

Return type `types.FunctionType`

`ckan.plugins.toolkit.get_or_bust(data_dict, keys)`

Return the value(s) from the given `data_dict` for the given key(s).

Usage:

```
single_value = get_or_bust(data_dict, 'a_key')
value_1, value_2 = get_or_bust(data_dict, ['key1', 'key2'])
```

Parameters

- **data_dict** (*dictionary*) – the dictionary to return the values from
- **keys** (*either a string or a list*) – the key(s) for the value(s) to return

Returns a single value from the dict if a single key was given, or a tuple of values if a list of keys was given

Raises `ckan.logic.ValidationError` if one of the given keys is not in the given dictionary

`ckan.plugins.toolkit.get_validator(validator)`

Return a validator function by name.

Parameters *validator* (*string*) – the name of the validator function to return, eg. `'package_name_exists'`

Raises `UnknownValidator` if the named validator is not found

Returns the named validator function

Return type `types.FunctionType`

class `ckan.plugins.toolkit.literal`

Represents an HTML literal.

This subclass of unicode has a `.__html__()` method that is detected by the `escape()` function.

Also, if you add another string to this string, the other string will be quoted and you will get back another literal object. Also `literal(...) % obj` will quote any value(s) from `obj`. If you do something like `literal(...) + literal(...)`, neither string will be changed because `escape(literal(...))` doesn't change the original literal.

Changed in WebHelpers 1.2: the implementation is now now a subclass of `markupsafe.Markup`. This brings some new methods: `.escape` (class method), `.unescape`, and `.striptags`.

`ckan.plugins.toolkit.missing`

`ckan.plugins.toolkit.navl_validate(data, schema, context=None)`

Validate an unflattened nested dict against a schema.

`ckan.plugins.toolkit.redirect_to(*args, **kw)`

Issue a redirect: return an HTTP response with a 302 Moved header.

This is a wrapper for `routes.redirect_to()` that maintains the user's selected language when redirecting.

The arguments to this function identify the route to redirect to, they're the same arguments as `ckan.plugins.toolkit.url_for()` accepts, for example:

```
import ckan.plugins.toolkit as toolkit

# Redirect to /dataset/my_dataset.
toolkit.redirect_to(controller='package', action='read',
                    id='my_dataset')
```

Or, using a named route:

```
toolkit.redirect_to('dataset_read', id='changed')
```

```
ckan.plugins.toolkit.render(template_name,          extra_vars=None,          cache_key=None,
                             cache_type=None,       cache_expire=None,       method='xhtml',
                             loader_class=<class    'genshi.template.markup.MarkupTemplate'>,
                             cache_force=None, renderer=None)
```

Render a template and return the output.

This is CKAN's main template rendering function.

Todo

Document the parameters of `ckan.plugins.toolkit.render()`.

```
ckan.plugins.toolkit.render_snippet(cls, template, data=None)
```

Render a template snippet and return the output.

See [Theming guide](#).

```
ckan.plugins.toolkit.render_text(template_name, extra_vars=None, cache_force=None)
```

Render a Genshi NewTextTemplate.

This is just a wrapper function that lets you render a Genshi NewTextTemplate without having to pass `method='text'` or `loader_class=NewTextTemplate` (it passes them to `render()` for you).

```
ckan.plugins.toolkit.request
```

The Pylons request object.

A new request object is created for each HTTP request. It has methods and attributes for getting things like the request headers, query-string variables, request body variables, cookies, the request URL, etc.

```
ckan.plugins.toolkit.requires_ckan_version(cls, min_version, max_version=None)
```

Raise `CkanVersionException` if the CKAN version is not greater than or equal to `min_version` and less then or equal to `max_version`.

If no `max_version` is given, just check whether the CKAN version is greater than or equal to `min_version`.

Plugins can call this function if they require a certain CKAN version, other versions of CKAN will crash if a user tries to use the plugin with them.

Parameters

- **min_version** (*string*) – the minimum acceptable CKAN version, eg. '2.1'
- **max_version** (*string*) – the maximum acceptable CKAN version, eg. '2.3'

```
ckan.plugins.toolkit.response
```

The Pylons response object.

Pylons uses this object to generate the HTTP response it returns to the web browser. It has attributes like the HTTP status code, the response headers, content type, cookies, etc.

```
ckan.plugins.toolkit.side_effect_free(action)
```

A decorator that marks the given action function as side-effect-free.

Action functions decorated with this decorator can be called with an HTTP GET request to the [Action API](#). Action functions that don't have this decorator must be called with a POST request.

If your CKAN extension defines its own action functions using the `IActions` plugin interface, you can use this decorator to make your actions available with GET requests instead of just with POST requests.

Example:

```
import ckan.plugins.toolkit as toolkit

@toolkit.side_effect_free
def my_custom_action_function(context, data_dict):
    ...
```

(Then implement `IActions` to register your action function with CKAN.)

```
ckan.plugins.toolkit.url_for(*args, **kw)
Return the URL for the given controller, action, id, etc.
```

Usage:

```
import ckan.plugins.toolkit as toolkit

url = toolkit.url_for(controller='package', action='read',
                      id='my_dataset')
=> returns '/dataset/my_dataset'
```

Or, using a named route:

```
toolkit.url_for('dataset_read', id='changed')
```

This is a wrapper for `routes.url_for()` that adds some extra features that CKAN needs.

Validator functions reference

```
ckan.logic.validators.owner_org_validator(key, data, errors, context)
```

```
ckan.logic.validators.package_id_not_changed(value, context)
```

```
ckan.logic.validators.int_validator(value, context)
```

Return an integer for value, which may be a string in base 10 or a numeric type (e.g. int, long, float, Decimal, Fraction). Return None for None or empty/all-whitespace string values.

Raises `ckan.lib.navl.dictization_functions.Invalid` for other inputs or non-whole values

```
ckan.logic.validators.natural_number_validator(value, context)
```

```
ckan.logic.validators.is_positive_integer(value, context)
```

```
ckan.logic.validators.boolean_validator(value, context)
```

Return a boolean for value. Return value when value is a python bool type. Return True for strings 'true', 'yes', 't', 'y', and '1'. Return False in all other cases, including when value is an empty string or None

```
ckan.logic.validators.isodate(value, context)
```

```
ckan.logic.validators.no_http(value, context)
```

```
ckan.logic.validators.package_id_exists(value, context)
```

```
ckan.logic.validators.package_id_does_not_exist(value, context)
```

```
ckan.logic.validators.package_name_exists(value, context)
```

`ckan.logic.validators.package_id_or_name_exists(package_id_or_name, context)`

Return the given `package_id_or_name` if such a package exists.

Raises `ckan.lib.navl.dictization_functions.Invalid` if there is no package with the given id or name

`ckan.logic.validators.user_id_exists(user_id, context)`

Raises `Invalid` if the given `user_id` does not exist in the model given in the context, otherwise returns the given `user_id`.

`ckan.logic.validators.user_id_or_name_exists(user_id_or_name, context)`

Return the given `user_id_or_name` if such a user exists.

Raises `ckan.lib.navl.dictization_functions.Invalid` if no user can be found with the given id or user name

`ckan.logic.validators.group_id_exists(group_id, context)`

Raises `Invalid` if the given `group_id` does not exist in the model given in the context, otherwise returns the given `group_id`.

`ckan.logic.validators.related_id_exists(related_id, context)`

Raises `Invalid` if the given `related_id` does not exist in the model given in the context, otherwise returns the given `related_id`.

`ckan.logic.validators.group_id_or_name_exists(reference, context)`

Raises `Invalid` if a group identified by the name or id cannot be found.

`ckan.logic.validators.activity_type_exists(activity_type)`

Raises `Invalid` if there is no registered activity renderer for the given `activity_type`. Otherwise returns the given `activity_type`.

This just uses `object_id_validators` as a lookup. very safe.

`ckan.logic.validators.resource_id_exists(value, context)`

`ckan.logic.validators.object_id_validator(key, activity_dict, errors, context)`

Validate the 'object_id' value of an activity_dict.

Uses the `object_id_validators` dict (above) to find and call an 'object_id' validator function for the given activity_dict's 'activity_type' value.

Raises `Invalid` if the model given in context contains no object of the correct type (according to the 'activity_type' value of the activity_dict) with the given ID.

Raises `Invalid` if there is no `object_id_validator` for the activity_dict's 'activity_type' value.

`ckan.logic.validators.name_validator(value, context)`

Return the given value if it's a valid name, otherwise raise `Invalid`.

If it's a valid name, the given value will be returned unmodified.

This function applies general validation rules for names of packages, groups, users, etc.

Most schemas also have their own custom name validator function to apply custom validation rules after this function, for example a `package_name_validator()` to check that no package with the given name already exists.

Raises `ckan.lib.navl.dictization_functions.Invalid` if `value` is not a valid name

`ckan.logic.validators.package_name_validator(key, data, errors, context)`

`ckan.logic.validators.package_version_validator(value, context)`

`ckan.logic.validators.duplicate_extras_key(key, data, errors, context)`

`ckan.logic.validators.group_name_validator(key, data, errors, context)`

`ckan.logic.validators.tag_length_validator` (*value, context*)

`ckan.logic.validators.tag_name_validator` (*value, context*)

`ckan.logic.validators.tag_not_uppercase` (*value, context*)

`ckan.logic.validators.tag_string_convert` (*key, data, errors, context*)
 Takes a list of tags that is a comma-separated string (in `data[key]`) and parses tag names. These are added to the data dict, enumerated. They are also validated.

`ckan.logic.validators.ignore_not_admin` (*key, data, errors, context*)

`ckan.logic.validators.ignore_not_package_admin` (*key, data, errors, context*)
 Ignore if the user is not allowed to administer the package specified.

`ckan.logic.validators.ignore_not_sysadmin` (*key, data, errors, context*)
 Ignore the field if user not sysadmin or `ignore_auth` in context.

`ckan.logic.validators.ignore_not_group_admin` (*key, data, errors, context*)
 Ignore if the user is not allowed to administer for the group specified.

`ckan.logic.validators.user_name_validator` (*key, data, errors, context*)
 Validate a new user name.

Append an error message to `errors[key]` if a user named `data[key]` already exists. Otherwise, do nothing.

Raises `ckan.lib.navl.dictization_functions.Invalid` if `data[key]` is not a string

Return type `None`

`ckan.logic.validators.user_both_passwords_entered` (*key, data, errors, context*)

`ckan.logic.validators.user_password_validator` (*key, data, errors, context*)

`ckan.logic.validators.user_passwords_match` (*key, data, errors, context*)

`ckan.logic.validators.user_password_not_empty` (*key, data, errors, context*)
 Only check if password is present if the user is created via action API. If not, `user_both_passwords_entered` will handle the validation

`ckan.logic.validators.user_about_validator` (*value, context*)

`ckan.logic.validators.vocabulary_name_validator` (*name, context*)

`ckan.logic.validators.vocabulary_id_not_changed` (*value, context*)

`ckan.logic.validators.vocabulary_id_exists` (*value, context*)

`ckan.logic.validators.tag_in_vocabulary_validator` (*value, context*)

`ckan.logic.validators.tag_not_in_vocabulary` (*key, tag_dict, errors, context*)

`ckan.logic.validators.url_validator` (*key, data, errors, context*)
 Checks that the provided value (if it is present) is a valid URL

`ckan.logic.validators.user_name_exists` (*user_name, context*)

`ckan.logic.validators.role_exists` (*role, context*)

`ckan.logic.validators.datasets_with_no_organization_cannot_be_private` (*key, data, errors, context*)

`ckan.logic.validators.list_of_strings` (*key, data, errors, context*)

`ckan.logic.validators.if_empty_guess_format` (*key, data, errors, context*)

`ckan.logic.validators.clean_format` (*format*)

`ckan.logic.validators.no_loops_in_hierarchy` (*key, data, errors, context*)

Checks that the parent groups specified in the data would not cause a loop in the group hierarchy, and therefore cause the recursion up/down the hierarchy to get into an infinite loop.

`ckan.logic.validators.filter_fields_and_values_should_have_same_length` (*key,*
data,
er-
rors,
con-
text)

`ckan.logic.validators.filter_fields_and_values_exist_and_are_valid` (*key, data,*
errors,
context)

`ckan.logic.validators.extra_key_not_in_root_schema` (*key, data, errors, context*)

`ckan.logic.validators.empty_if_not_sysadmin` (*key, data, errors, context*)

Only sysadmins may pass this value

`ckan.logic.converters.convert_to_extras` (*key, data, errors, context*)

`ckan.logic.converters.convert_from_extras` (*key, data, errors, context*)

`ckan.logic.converters.extras_unicode_convert` (*extras, context*)

`ckan.logic.converters.free_tags_only` (*key, data, errors, context*)

`ckan.logic.converters.convert_to_tags` (*vocab*)

`ckan.logic.converters.convert_from_tags` (*vocab*)

`ckan.logic.converters.convert_user_name_or_id_to_id` (*user_name_or_id, context*)

Return the user id for the given user name or id.

The point of this function is to convert user names to ids. If you have something that may be a user name or a user id you can pass it into this function and get the user id out either way.

Also validates that a user with the given name or id exists.

Returns the id of the user with the given user name or id

Return type string

Raises `ckan.lib.navl.dictization_functions.Invalid` if no user can be found with the given id or user name

`ckan.logic.converters.convert_package_name_or_id_to_id` (*package_name_or_id, con-*
text)

Return the package id for the given package name or id.

The point of this function is to convert package names to ids. If you have something that may be a package name or id you can pass it into this function and get the id out either way.

Also validates that a package with the given name or id exists.

Returns the id of the package with the given name or id

Return type string

Raises `ckan.lib.navl.dictization_functions.Invalid` if there is no package with the given name or id

`ckan.logic.converters.convert_group_name_or_id_to_id(group_name_or_id, context)`

Return the group id for the given group name or id.

The point of this function is to convert group names to ids. If you have something that may be a group name or id you can pass it into this function and get the id out either way.

Also validates that a group with the given name or id exists.

Returns the id of the group with the given name or id

Return type string

Raises `ckan.lib.navl.dictization_functions.Invalid` if there is no group with the given name or id

`ckan.logic.converters.convert_to_json_if_string(value, context)`

`ckan.logic.converters.convert_to_list_if_string(value, context=None)`

`ckan.logic.converters.remove_whitespace(value, context)`

Theming guide

Changed in version 2.0: The CKAN frontend was completely rewritten for CKAN 2.0, so most of the tutorials below don't apply to earlier versions of CKAN.

The following sections will teach you how to customize the content and appearance of CKAN pages by developing your own CKAN themes.

See also:

Getting started If you just want to do some simple customizations such as changing the title of your CKAN site, or making some small CSS customizations, *Getting started* documents some simple configuration settings you can use.

Note: Before you can start developing a CKAN theme, you'll need a working source install of CKAN on your system. If you don't have a CKAN source install already, follow the instructions in *Installing CKAN from source* before continuing.

Note: CKAN theme development is a technical topic, for web developers. The tutorials below assume basic knowledge of:

- [The Python programming language](#)
- [HTML](#)
- [CSS](#)
- [JavaScript](#)

We also recommend familiarizing yourself with:

- [Jinja2 templates](#)
 - [Bootstrap](#)
 - [jQuery](#)
-

Customizing CKAN's templates

CKAN pages are generated from [Jinja2](#) template files. This tutorial will walk you through the process of writing your own template files to modify and replace the default ones, and change the layout and content of CKAN pages.

See also:

String internationalization How to mark strings for translation in your template files.

Creating a CKAN extension

A CKAN theme is simply a CKAN plugin that contains some custom templates and static files, so before getting started on our CKAN theme we'll have to create an extension and plugin. For a detailed explanation of the steps below, see *Writing extensions tutorial*.

1. Use the `paster create` command to create an empty extension:

```
. /usr/lib/ckan/default/bin/activate
cd /usr/lib/ckan/default/src
paster --plugin=ckan create -t ckanext ckanext-example_theme
```

2. Create the file `ckanext-example_theme/ckanext/example_theme/plugin.py` with the following contents:

```
import ckan.plugins as plugins

class ExampleThemePlugin(plugins.SingletonPlugin):
    '''An example theme plugin.

    '''
    pass
```

3. Edit the entry points in `ckanext-example_theme/setup.py` to look like this:

```
entry_points='''
    [ckan.plugins]
    example_theme=ckanext.example_theme.plugin:ExampleThemePlugin
''',
```

4. Run `python setup.py develop`:

```
cd ckanext-example_theme
python setup.py develop
```

5. Add the plugin to the `ckan.plugins` setting in your `/etc/ckan/default/development.ini` file:

```
ckan.plugins = stats text_view recline_view example_theme
```

6. Start CKAN in the development web server:

```
$ paster serve --reload /etc/ckan/default/development.ini
Starting server in PID 13961.
serving on 0.0.0.0:5000 view at http://127.0.0.1:5000
```

Open the [CKAN front page](#) in your web browser. If your plugin is in the `ckan.plugins` setting and CKAN starts without crashing, then your plugin is installed and CKAN can find it. Of course, your plugin doesn't *do* anything yet.

Replacing a default template file

Every CKAN page is generated by rendering a particular template. For each page of a CKAN site there's a corresponding template file. For example the front page is generated from the `ckan/templates/home/index.html` file, the `/about` page is generated from `ckan/templates/home/about.html`, the datasets page at `/dataset` is generated from `ckan/templates/package/search.html`, etc.

To customize pages, our plugin needs to register its own custom template directory containing template files that override the default ones. Edit the `ckanext-example_theme/ckanext/example_theme/plugin.py` file that we created earlier, so that it looks like this:

```
'''plugin.py

'''
import ckan.plugins as plugins
import ckan.plugins.toolkit as toolkit

class ExampleThemePlugin(plugins.SingletonPlugin):
    '''An example theme plugin.

    '''
    # Declare that this class implements IConfigurer.
    plugins.implements(plugins.IConfigurer)

    def update_config(self, config):

        # Add this plugin's templates dir to CKAN's extra_template_paths, so
        # that CKAN will use this plugin's custom templates.
        # 'templates' is the path to the templates dir, relative to this
        # plugin.py file.
        toolkit.add_template_directory(config, 'templates')
```

This new code does a few things:

1. It imports CKAN's *plugins toolkit* module:

```
import ckan.plugins.toolkit as toolkit
```

The plugins toolkit is a Python module containing core functions, classes and exceptions for CKAN plugins to use. For more about the plugins toolkit, see [Writing extensions tutorial](#).

2. It calls `implements()` to declare that it implements the `IConfigurer` plugin interface:

```
plugins.implements(plugins.IConfigurer)
```

This tells CKAN that our `ExampleThemePlugin` class implements the methods declared in the `IConfigurer` interface. CKAN will call these methods of our plugin class at the appropriate times.

3. It implements the `update_config()` method, which is the only method declared in the `IConfigurer` interface:

```
def update_config(self, config):

    # Add this plugin's templates dir to CKAN's extra_template_paths, so
    # that CKAN will use this plugin's custom templates.
    # 'templates' is the path to the templates dir, relative to this
    # plugin.py file.
    toolkit.add_template_directory(config, 'templates')
```

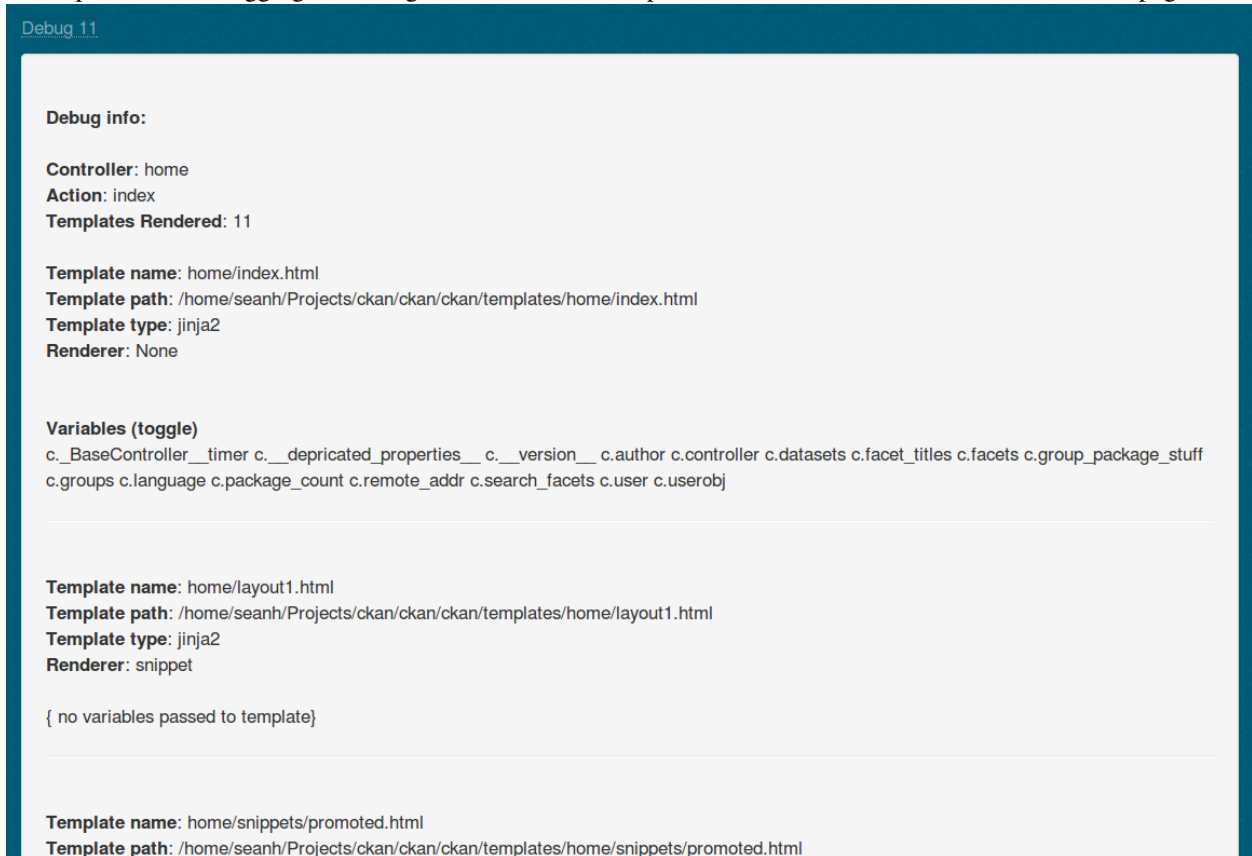
CKAN will call this method when it starts up, to give our plugin a chance to modify CKAN's configuration settings. Our `update_config()` method calls `add_template_directory()` to register its custom template directory with CKAN. This tells CKAN to look for template files in `ckanext-example_theme/ckanext/example_theme/templates` whenever it renders a page. Any template file in this directory that has the same name as one of CKAN's default template files, will be used instead of the default file.

Now, let's customize the CKAN front page. We first need to discover which template file CKAN uses to render the front page, so we can replace it. Set *debug* to `true` in your `/etc/ckan/default/development.ini` file:

[DEFAULT]

```
# WARNING: *THIS SETTING MUST BE SET TO FALSE ON A PRODUCTION ENVIRONMENT*
debug = true
```

Reload the [CKAN front page](#) in your browser, and you should see a *Debug* link in the footer at the bottom of the page. Click on this link to open the debug footer. The debug footer displays various information useful for CKAN frontend development and debugging, including the names of the template files that were used to render the current page:



The screenshot shows the 'Debug 11' footer. It contains the following information:

- Debug info:**
 - Controller: home
 - Action: index
 - Templates Rendered: 11
- Template name:** home/index.html
- Template path:** /home/seanh/Projects/ckan/ckan/ckan/templates/home/index.html
- Template type:** jinja2
- Renderer:** None
- Variables (toggle):**
 - c._BaseController__timer c._depricated_properties__ c._version__ c.author c.controller c.datasets c.facet_titles c.facets c.group_package_stuff c.groups c.language c.package_count c.remote_addr c.search_facets c.user c.userobj
- Template name:** home/layout1.html
- Template path:** /home/seanh/Projects/ckan/ckan/ckan/templates/home/layout1.html
- Template type:** jinja2
- Renderer:** snippet
- { no variables passed to template }
- Template name:** home/snippets/promoted.html
- Template path:** /home/seanh/Projects/ckan/ckan/ckan/templates/home/snippets/promoted.html

The first template file listed is the one we're interested in:

```
Template name: home/index.html
Template path: /usr/lib/ckan/default/src/ckan/ckan/templates/home/index.html
```

This tells us that `home/index.html` is the root template file used to render the front page. The debug footer appears at the bottom of every CKAN page, and can always be used to find the page's template files, and other information about the page.

Note: Most CKAN pages are rendered from multiple template files. The first file listed in the debug footer is the root template file of the page. All other template files used to render the page (listed further down in the debug footer) are either included by the root file, or included by another file that is included by the root file.

To figure out which template file renders a particular part of the page you have to inspect the [source code of the template files](#), starting with the root file.

Now let's override `home/index.html` using our plugins' custom templates directory. Create the `ckanext-example_theme/ckanext/example_theme/templates` directory, create a `home` directory inside the `templates` directory, and create an empty `index.html` file inside the `home` directory:

```
ckanext-example_theme/
  ckanext/
    example_theme/
      templates/
        home/
          index.html <-- An empty file.
```

If you now restart the development web server (kill the server using Ctrl-c, then run the `paster serve` command again) and reload the [CKAN front page](#) in your web browser, you should see an empty page, because we've replaced the template file for the front page with an empty file.

Note: If you run `paster serve` with the `--reload` option, then it isn't usually necessary to restart the server after editing a Python file, a template file, your CKAN config file, or any other CKAN file. If you've added a new file or directory, however, you need to restart the server manually.

Jinja2

CKAN template files are written in the [Jinja2](#) templating language. Jinja template files, such as our `index.html` file, are simply text files that, when processed, generate any text-based output format such as HTML, XML, CSV, etc. Most of the template files in CKAN generate HTML.

We'll introduce some Jinja2 basics below. Jinja2 templates have many more features than these, for full details see the [Jinja2 docs](#).

Expressions and variables

Jinja2 *expressions* are snippets of code between `{{ ... }}` delimiters, when a template is rendered any expressions are evaluated and replaced with the resulting value.

The simplest use of an expression is to display the value of a variable, for example `{{ foo }}` in a template file will be replaced with the value of the variable `foo` when the template is rendered.

CKAN makes a number of global variables available to all templates. One such variable is `app_globals`, which can be used to access certain global attributes including some of the settings from your CKAN config file. For example, to display the value of the `ckan.site_title` setting from your config file you would put this code in any template file:

```
<p>The title of this site is: {{ app_globals.site_title }}.</p>
```

Note: The `app_globals` variable is also sometimes called `g` (an alias), you may see `g` in some CKAN templates. See [Variables and functions available to templates](#).

Note: Not all config settings are available to templates via `app_globals`. The `sqlalchemy.url` setting, for example, contains your database password, so making that variable available to templates might be a security risk.

If you've added your own custom options to your config file, these will not be available in `app_globals` automatically. See [Accessing custom config settings from templates](#).

Note: If a template tries to render a variable or attribute that doesn't exist, rather than crashing or giving an error message, the Jinja2 expression simply evaluates to nothing (an empty string). For example, these Jinja2 expressions will output nothing:

```
{{ app_globals.an_attribute_that_does_not_exist }}
```

```
{{ a_variable_that_does_not_exist }}
```

If, on the other hand, you try to render an attribute of a variable that doesn't exist, then Jinja2 will crash. For example, this Jinja2 expression will crash with an `UndefinedError`: `'a_variable_that_does_not_exist'` is undefined:

```
{{ a_variable_that_does_not_exist.an_attribute_that_does_not_exist }}
```

See the [Jinja2 variables docs](#) for details.

Note: Jinja2 expressions can do much more than print out the values of variables, for example they can call Jinja2's [global functions](#), CKAN's [template helper functions](#) and any [custom template helper functions](#) provided by your extension, and use any of the [literals and operators](#) that Jinja provides.

See [Variables and functions available to templates](#) for a list of variables and functions available to templates.

Tags

`ckan.site_title` is an example of a simple string variable. Some variables, such as `ckan.plugins`, are lists, and can be looped over using Jinja's `{% for %}` tag.

Jinja *tags* are snippets of code between `{% ... %}` delimiters that control the logic of the template. For example, we can output a list of the currently enabled plugins with this code in any template file:

```
<p>The currently enabled plugins are:</p>
<ul>
  {% for plugin in app_globals.plugins %}
    <li>{{ plugin }}</li>
  {% endfor %}
</ul>
```

Other variables, such as `ckan.tracking_enabled`, are booleans, and can be tested using Jinja's `{% if %}` tag:

```
{% if g.tracking_enabled %}
  <p>CKAN's page-view tracking feature is enabled.</p>
{% else %}
  <p>CKAN's page-view tracking feature is <i>not</i> enabled.</p>
{% endif %}
```

Comments

Finally, any text between `{# ... #}` delimiters in a Jinja2 template is a *comment*, and will not be output when the template is rendered:

```
{# This text will not appear in the output when this template is rendered. #}
```

Extending templates with `{% ckan_extends %}`

CKAN provides a custom Jinja tag `{% ckan_extends %}` that we can use to declare that our `home/index.html` template extends the default `home/index.html` template, instead of completely replacing it. Edit the empty `index.html` file you just created, and add one line:

```
{% ckan_extends %}
```

If you now reload the [CKAN front page](#) in your browser, you should see the normal front page appear again. When CKAN processes our `index.html` file, the `{% ckan_extends %}` tag tells it to process the default `home/index.html` file first.

Replacing template blocks with `{% block %}`

Jinja templates can contain *blocks* that child templates can override. For example, CKAN’s default `home/layout1.html` template (one of the files used to render the CKAN front page) has a block that contains the Jinja and HTML code for the “featured group” that appears on the front page:

```
{% block featured_group %}
    {% snippet 'home/snippets/featured_group.html' %}
{% endblock %}
```

Note: This code calls a *template snippet* that contains the actual Jinja and HTML code for the featured group, more on snippets later.

Note: The CKAN front page supports a number of different layouts: `layout1`, `layout2`, `layout3`, etc. The layout can be chosen by a sysadmin using the *admin page*. This tutorial assumes your CKAN is set to use the first (default) layout.

When a custom template file extends one of CKAN’s default template files using `{% ckan_extends %}`, it can replace any of the blocks from the default template with its own code by using `{% block %}`. Create the file `ckanext-example_theme/ckanext/example_theme/templates/home/layout1.html` with these contents:

```
{% ckan_extends %}

{% block featured_group %}
    Hello block world!
{% endblock %}
```

This file extends the default `layout1.html` template, and overrides the `featured_group` block. Restart the development web server and reload the [CKAN front page](#) in your browser. You should see that the featured groups section of the page has been replaced, but the rest of the page remains intact.

Note: Most template files in CKAN contain multiple blocks. To find out what blocks a template has, and which block renders a particular part of the page, you have to look at the [source code of the default template files](#).

Extending parent blocks with Jinja’s `{{ super() }}`

If you want to add some code to a block but don’t want to replace the entire block, you can use Jinja’s `{{ super() }}` tag:

```
{% ckan_extends %}

{% block featured_group %}

    <p>This paragraph will be added to the top of the
    <code>featured_group</code> block.</p>

    {# Insert the contents of the original featured_group block: #}
    {{ super() }}

    <p>This paragraph will be added to the bottom of the
    <code>featured_group</code> block.</p>

{% endblock %}
```

When the child block above is rendered, Jinja will replace the `{{ super() }}` tag with the contents of the parent block. The `{{ super() }}` tag can be placed anywhere in the block.

Template helper functions

Now let's put some interesting content into our custom template block. One way for templates to get content out of CKAN is by calling CKAN's *template helper functions*.

For example, let's replace the featured group on the front page with an activity stream of the site's recently created, updated and deleted datasets. Change the code in `ckanext-example_theme/ckanext/example_theme/templates/home/layout1.html` to this:

```
{% ckan_extends %}

{% block featured_group %}
    {{ h.recently_changed_packages_activity_stream(limit=4) }}
{% endblock %}
```

Reload the CKAN front page in your browser and you should see a new activity stream:



To call a template helper function we use a Jinja2 *expression* (code wrapped in `{{ ... }}` brackets), and we use the global variable `h` (available to all templates) to access the helper:

```
{{ h.recently_changed_packages_activity_stream(limit=4) }}
```

To see what other template helper functions are available, look at the *template helper functions reference docs*.

Adding your own template helper functions

Plugins can add their own template helper functions by implementing CKAN's `ITemplateHelpers` plugin interface. (see *Writing extensions tutorial* for a detailed explanation of CKAN plugins and plugin interfaces).

Let's add another item to our custom front page: a list of the most "popular" groups on the site (the groups with the most datasets). We'll add a custom template helper function to select the groups to be shown. First, in our `plugin.py` file

we need to implement `ITemplateHelpers` and provide our helper function. Change the contents of `plugin.py` to look like this:

```
import ckan.plugins as plugins
import ckan.plugins.toolkit as toolkit

def most_popular_groups():
    '''Return a sorted list of the groups with the most datasets.'''

    # Get a list of all the site's groups from CKAN, sorted by number of
    # datasets.
    groups = toolkit.get_action('group_list')(
        data_dict={'sort': 'packages desc', 'all_fields': True})

    # Truncate the list to the 10 most popular groups only.
    groups = groups[:10]

    return groups

class ExampleThemePlugin(plugins.SingletonPlugin):
    '''An example theme plugin.

    '''
    plugins.implements(plugins.IConfigurer)

    # Declare that this plugin will implement ITemplateHelpers.
    plugins.implements(plugins.ITemplateHelpers)

    def update_config(self, config):

        # Add this plugin's templates dir to CKAN's extra_template_paths, so
        # that CKAN will use this plugin's custom templates.
        toolkit.add_template_directory(config, 'templates')

    def get_helpers(self):
        '''Register the most_popular_groups() function above as a template
        helper function.

        '''
        # Template helper function names should begin with the name of the
        # extension they belong to, to avoid clashing with functions from
        # other extensions.
        return {'example_theme_most_popular_groups': most_popular_groups}
```

We've added a number of new features to `plugin.py`. First, we defined a function to get the most popular groups from CKAN:

```
def most_popular_groups():
    '''Return a sorted list of the groups with the most datasets.'''

    # Get a list of all the site's groups from CKAN, sorted by number of
    # datasets.
    groups = toolkit.get_action('group_list')(
        data_dict={'sort': 'packages desc', 'all_fields': True})

    # Truncate the list to the 10 most popular groups only.
    groups = groups[:10]
```

```
return groups
```

This function calls one of CKAN's *action functions* to get the groups from CKAN. See [Writing extensions tutorial](#) for more about action functions.

Next, we called `implements()` to declare that our class now implements `ITemplateHelpers`:

```
plugins.implements(plugins.ITemplateHelpers)
```

Finally, we implemented the `get_helpers()` method from `ITemplateHelpers` to register our function as a template helper:

```
def get_helpers(self):
    '''Register the most_popular_groups() function above as a template
    helper function.

    '''
    # Template helper function names should begin with the name of the
    # extension they belong to, to avoid clashing with functions from
    # other extensions.
    return {'example_theme_most_popular_groups': most_popular_groups}
```

Now that we've registered our helper function, we need to call it from our template. As with CKAN's default template helpers, templates access custom helpers via the global variable `h`. Edit `ckanext-example_theme/ckanext/example_theme/templates/home/layout1.html` to look like this:

```
{% ckan_extends %}

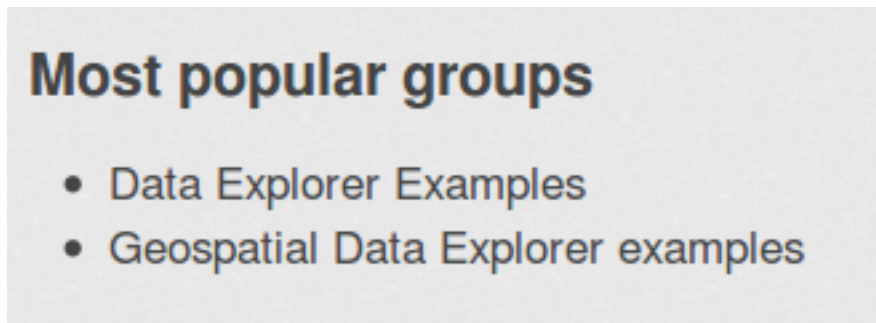
{% block featured_group %}
    {{ h.recently_changed_packages_activity_stream(limit=4) }}
{% endblock %}

{% block featured_organization %}

    {# Show a list of the site's most popular groups. #}
    <h3>Most popular groups</h3>
    <ul>
        {% for group in h.example_theme_most_popular_groups() %}
            <li>{{ group.display_name }}</li>
        {% endfor %}
    </ul>

{% endblock %}
```

Now reload your CKAN front page in your browser. You should see the featured organization section replaced with a list of the most popular groups:



Simply displaying a list of group titles isn't very good. We want the groups to be hyperlinked to their pages, and also to show some other information about the group such as its description and logo image. To display our groups nicely, we'll use CKAN's *template snippets*...

Template snippets

Template snippets are small snippets of template code that, just like helper functions, can be called from any template file. To call a snippet, you use another of CKAN's custom Jinja2 tags: `{% snippet %}`. CKAN comes with a selection of snippets, which you can find in the various `snippets` directories in `ckan/templates/`, such as `ckan/templates/snippets/` and `ckan/templates/package/snippets/`. For a complete list of the default snippets available to templates, see *Template snippets reference*.

`ckan/templates/group/snippets/group_list.html` is a snippet that renders a list of groups nicely (it's used to render the groups on CKAN's `/group` page and on user dashboard pages, for example):

```
{#
Display a grid of group items.

groups - A list of groups.

Example:

    {% snippet "group/snippets/group_list.html" %}

#}
{% block group_list %}
<ul class="media-grid" data-module="media-grid">
    {% block group_list_inner %}
    {% for group in groups %}
        {% snippet "group/snippets/group_item.html", group=group, position=loop.index %}
    {% endfor %}
    {% endblock %}
</ul>
{% endblock %}
```

(As you can see, this snippet calls another snippet, `group_item.html`, to render each individual group.)

Let's change our `ckanext-example_theme/ckanext/example_theme/templates/home/layout1.html` file to call this snippet:

```
{% ckan_extends %}

{% block featured_group %}

    {{ h.recently_changed_packages_activity_stream(limit=4) }}

{% endblock %}

{% block featured_organization %}

<h3>Most popular groups</h3>

    {# Call the group_list.html snippet. #}
    {% snippet 'group/snippets/group_list.html',
        groups=h.example_theme_most_popular_groups() %}

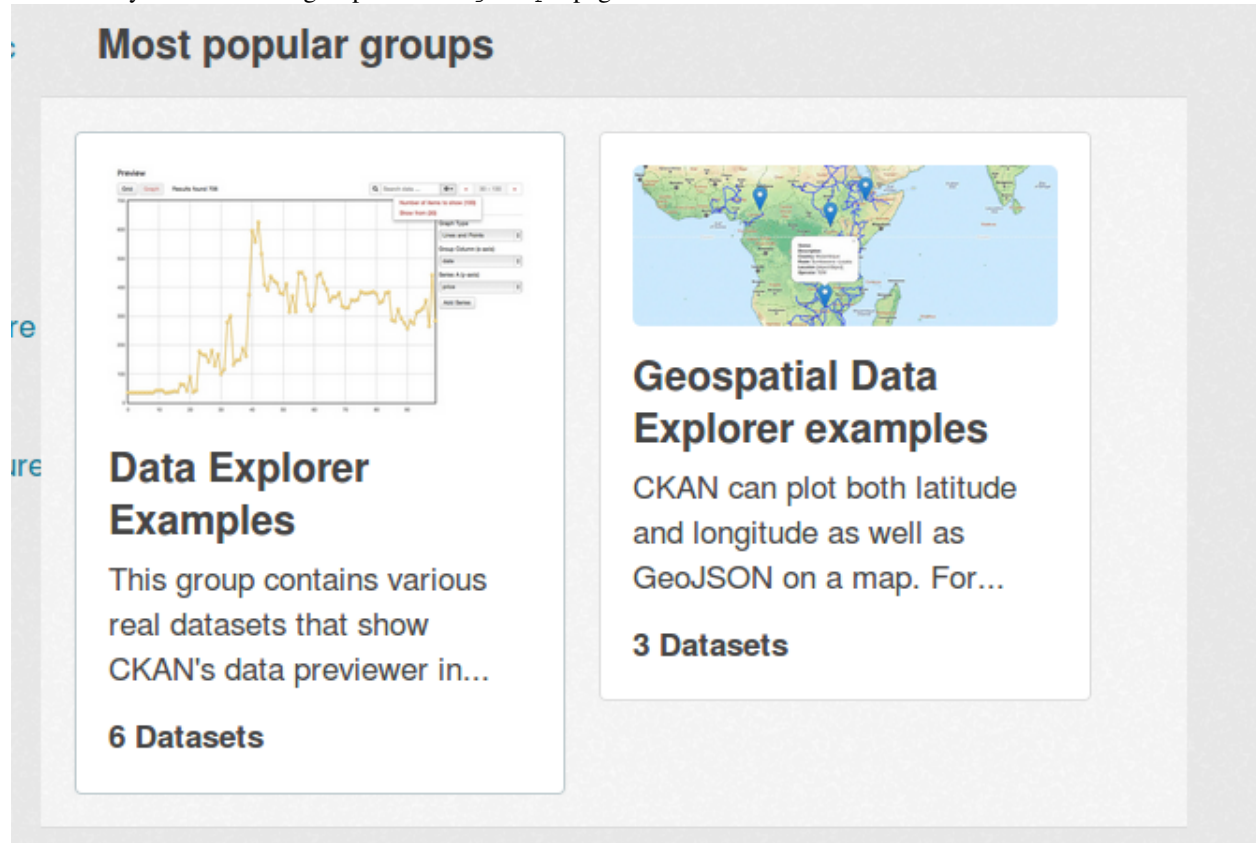
{% endblock %}
```

Here we pass two arguments to the `{% snippet %}` tag:

```
{% snippet 'group/snippets/group_list.html',
    groups=h.example_theme_most_popular_groups() %}
```

the first argument is the name of the snippet file to call. The second argument, separated by a comma, is the list of groups to pass into the snippet. After the filename you can pass any number of variables into a snippet, and these will all be available to the snippet code as top-level global variables. As in the `group_list.html` docstring above, each snippet's docstring should document the parameters it requires.

If you reload your [CKAN front page](#) in your web browser now, you should see the most popular groups rendered in the same style as the list of groups on the `/groups` page:



This style isn't really what we want for our front page, each group is too big. To render the groups in a custom style, we can define a custom snippet...

Adding your own template snippets

Just as plugins can add their own template helper functions, they can also add their own snippets. To add template snippets, all a plugin needs to do is add a `snippets` directory in its `templates` directory, and start adding files. The snippets will be callable from other templates immediately.

Note: For CKAN to find your plugins' snippets directories, you should already have added your plugin's custom template directory to CKAN, see [Replacing a default template file](#).

Let's create a custom snippet to display our most popular groups, we'll put the `<h3>Most popular groups</h3>` heading into the snippet and make it nice and modular, so that we can reuse the whole thing on different parts of the site if we want to.

Create a new directory `ckanext-example_theme/ckanext/example_theme/templates/snippets` containing a file named `example_theme_most_popular_groups.html` with these contents:

```
{#
Renders a list of the site's most popular groups.

groups - the list of groups to render

#}
<h3>Most popular groups</h3>
<ul>
  {% for group in groups %}
    <li>
      <a href="{{ h.url_for('group_read', action='read', id=group.name) }}">
        <h3>{{ group.display_name }}</h3>
      </a>
      {% if group.description %}
        <p>
          {{ h.markdown_extract(group.description, extract_length=80) }}
        </p>
      {% else %}
        <p>{{ _('This group has no description') }}</p>
      {% endif %}
      {% if group.packages %}
        <strong>{{ ungettext('{num} Dataset', '{num} Datasets', group.packages).format(num=group.pac
      {% else %}
        <span>{{ _('0 Datasets') }}</span>
      {% endif %}
    </li>
  {% endfor %}
</ul>
```

Note: As in the example above, a snippet should have a docstring at the top of the file that briefly documents what the snippet does and what parameters it requires. See *Snippets should have docstrings*.

This code uses a Jinja2 `for` loop to render each of the groups, and calls a number of CKAN's template helper functions:

- To hyperlink each group's name to the group's page, it calls `url_for()`.
- If the group has a description, it calls `markdown_extract()` to render the description nicely.
- If the group doesn't have a description, it uses the `_()` function to mark the 'This group has no description' message for translation. When the page is rendered in a user's web browser, this string will be shown in the user's language (if there's a translation of the string into that language).
- When rendering the group's number of datasets, it uses the `ungettext()` function to mark the message for translation with localized handling of plural forms.

The code also accesses the attributes of each group: `{{ group.name }}`, `{{ group.display_name }}`, `{{ group.description }}`, `{{ group.packages }}`, etc. To see what attributes a group or any other CKAN object (packages/datasets, organizations, users...) has, you can use [CKAN's API](#) to inspect the object. For example to find out what attributes a group has, call the `group_show()` function.

Now edit your `ckanext-example_theme/ckanext/example_theme/templates/home/layout1.html` file and change it to use our new snippet instead of the default one:


```
{% ckan_extends %}

{% block featured_group %}
    {{ h.recently_changed_packages_activity_stream(limit=4) }}
{% endblock %}

{% block featured_organization %}
    {% snippet 'snippets/example_theme_most_popular_groups.html',
              groups=h.example_theme_most_popular_groups() %}
{% endblock %}
```

Restart the development web server and reload the CKAN front page and you should see the most popular groups rendered differently:

Most popular groups

-

Data Explorer Examples

This group contains various real datasets that show CKAN's data previewer in...

6 Datasets

-

Geospatial Data Explorer examples

CKAN can plot both latitude and longitude as well as GeoJSON on a map. For...

3 Datasets

Warning: Default snippets can be overridden. If a plugin adds a snippet with the same name as one of CKAN's default snippets, the plugin's snippet will override the default snippet wherever the default snippet is used. Also if two plugins both have snippets with the same name, one of the snippets will override the other. To avoid unintended conflicts, we recommend that snippet filenames begin with the name of the extension they belong to, e.g. `snippets/example_theme_*.html`. See *Snippet filenames should begin with the name of the extension*.

Note: Snippets don't have access to the global template context variable, `c` (see *Variables and functions available to templates*). Snippets *can* access other global variables such as `h`, `app_globals` and `request`, as well as any variables explicitly passed into the snippet by the parent template when it calls the snippet with a `{% snippet %}` tag.

HTML tags and CSS classes

Our additions to the front page so far don't look very good or fit in very well with the CKAN theme. Let's make them look better by tweaking our template to use the right HTML tags and CSS classes.

There are two places to look for CSS classes available in CKAN:

1. The Bootstrap 2.3.2 docs. All of the HTML, CSS and JavaScript provided by Bootstrap is available to use in CKAN.
2. CKAN's development primer page, which can be found on any CKAN site at `/development/primer.html`, for example demo.ckan.org/development/primer.html.

The primer page demonstrates many of the HTML and CSS elements available in CKAN, and by viewing the source of the page you can see what HTML tags and CSS classes they use.

Edit your `example_theme_most_popular_groups.html` file to look like this:

```
{# Renders a list of the site's most popular groups. #}

<div class="box">
  <header class="module-heading">
    <h3>Most popular groups</h3>
  </header>
  <section class="module-content">
    <ul class="unstyled">
      {% for group in h.example_theme_most_popular_groups() %}
        <li>
          <a href="{{ h.url_for('group_read', action='read', id=group.name) }}">
            <h3>{{ group.display_name }}</h3>
          </a>
          {% if group.description %}
            <p>
              {{ h.markdown_extract(group.description, extract_length=80) }}
            </p>
          {% else %}
            <p>{{ _('This group has no description') }}</p>
          {% endif %}
          {% if group.packages %}
            <strong>{{ ungettext('{num} Dataset', '{num} Datasets', group.packages).format(num=group.packages) }}</strong>
          {% else %}
            <span>{{ _('0 Datasets') }}</span>
          {% endif %}
        </li>
      {% endfor %}
    </ul>
  </section>
</div>
```

This simply wraps the code in a `<div class="box">`, a `<header class="module-heading">`, and a `<section class="module-content">`. We also added Bootstrap's `class="unstyled"` to the `` tag to get rid of the bullet points. If you reload the [CKAN front page](#), the most popular groups should look much better.

To wrap your activity stream in a similar box, edit `layout1.html` to look like this:

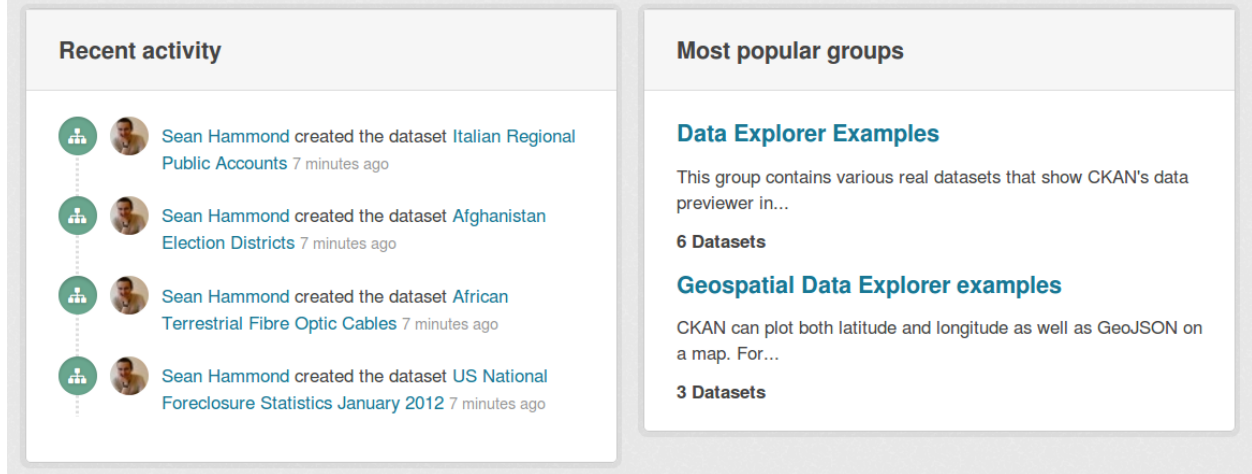
```
{% ckan_extends %}

{% block featured_group %}
  <div class="box">
    <header class="module-heading">
      <h3>Recent activity</h3>
```

```
</header>
<div class="module-content">
    {{ h.recently_changed_packages_activity_stream(limit=4) }}
</div>
</div>
{% endblock %}

{% block featured_organization %}
    {% snippet 'snippets/example_theme_most_popular_groups.html' %}
{% endblock %}
```

Reload the CKAN front page, and you should see your activity stream and most popular groups looking much better:



Accessing custom config settings from templates

Not all CKAN config settings are available to templates via `app_globals`. In particular, if an extension wants to use its own custom config setting, this setting will not be available. If you need to access a custom config setting from a template, you can do so by wrapping the config setting in a helper function.

See also:

For more on custom config settings, see *Using custom config settings in extensions*.

Todo

I'm not sure if making config settings available to templates like this is a very good idea. Is there an alternative best practice?

Let's add a config setting, `show_most_popular_groups`, to enable or disable the most popular groups on the front page. First, add a new helper function to `plugin.py` to wrap the config setting.

```
import pylons.config as config

import ckan.plugins as plugins
import ckan.plugins.toolkit as toolkit

def show_most_popular_groups():
    '''Return the value of the most_popular_groups config setting.

    To enable showing the most popular groups, add this line to the
```

```

[app:main] section of your CKAN config file::

    ckan.example_theme.show_most_popular_groups = True

Returns ``False`` by default, if the setting is not in the config file.

:rtype: boolean

'''
value = config.get('ckan.example_theme.show_most_popular_groups', False)
value = toolkit.asbool(value)
return value

def most_popular_groups():
    '''Return a sorted list of the groups with the most datasets.'''

    # Get a list of all the site's groups from CKAN, sorted by number of
    # datasets.
    groups = toolkit.get_action('group_list')(
        data_dict={'sort': 'packages desc', 'all_fields': True})

    # Truncate the list to the 10 most popular groups only.
    groups = groups[:10]

    return groups

class ExampleThemePlugin(plugins.SingletonPlugin):
    '''An example theme plugin.

    '''
    plugins.implements(plugins.IConfigurer)

    # Declare that this plugin will implement ITemplateHelpers.
    plugins.implements(plugins.ITemplateHelpers)

    def update_config(self, config):

        # Add this plugin's templates dir to CKAN's extra_template_paths, so
        # that CKAN will use this plugin's custom templates.
        toolkit.add_template_directory(config, 'templates')

    def get_helpers(self):
        '''Register the most_popular_groups() function above as a template
        helper function.

        '''
        # Template helper function names should begin with the name of the
        # extension they belong to, to avoid clashing with functions from
        # other extensions.
        return {'example_theme_most_popular_groups': most_popular_groups,
                'example_theme_show_most_popular_groups':
                show_most_popular_groups,
                }

def show_most_popular_groups():
    '''Return the value of the most_popular_groups config setting.

```

To enable showing the most popular groups, add this line to the `[app:main]` section of your CKAN config file::

```
ckan.example_theme.show_most_popular_groups = True
```

Returns `False` by default, if the setting is not in the config file.

```
:rtype: boolean
```

```
'''
```

```
value = config.get('ckan.example_theme.show_most_popular_groups', False)
```

```
value = toolkit.asbool(value)
```

```
return value
```

Note: Names of config settings provided by extensions should include the name of the extension, to avoid conflicting with core config settings or with config settings from other extensions. See *Names of config settings should include the name of the extension*.

Now we can call this helper function from our `layout1.html` template:

```
{% block featured_organization %}
  {% if h.example_theme_show_most_popular_groups() %}
    {% snippet 'snippets/example_theme_most_popular_groups.html' %}
  {% else %}
    {{ super() }}
  {% endif %}
{% endblock %}
```

If the user sets this config setting to `True` in their CKAN config file, then the most popular groups will be displayed on the front page, otherwise the block will fall back to its default contents.

Adding static files

You may need to add some custom *static files* to your CKAN site and use them from your templates, for example image files, PDF files, or any other static files that should be returned as-is by the webserver (as opposed to Jinja template files, which CKAN renders before returning them to the user).

By adding a directory to CKAN's *extra_public_paths* config setting, a plugin can make a directory of static files available to be used or linked to by templates. Let's add a static image file, and change the home page template to use our file as the promoted image on the front page.

See also:

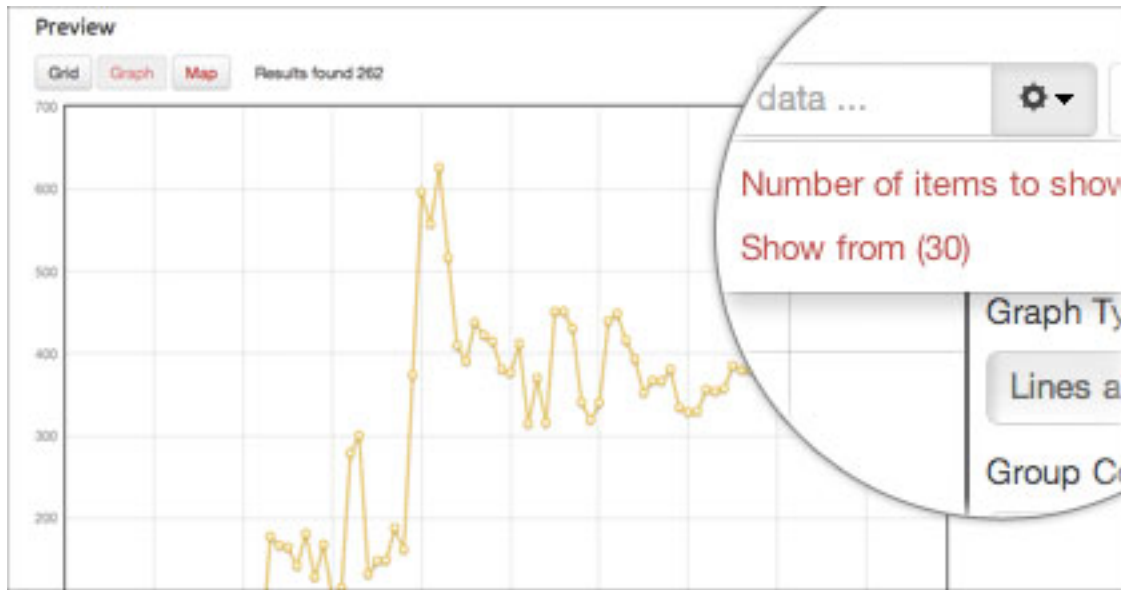
Adding CSS and JavaScript files using Fanstatic

If you're adding CSS files consider using Fanstatic instead of *extra_public_paths*, to take advantage of extra features. See *Adding CSS and JavaScript files using Fanstatic*. If you're adding JavaScript modules you have to use Fanstatic, see *Customizing CKAN's JavaScript*.

First, create a public directory in your extension with a `promoted-image.jpg` file in it:

```
ckanext-example_theme/
  ckanext/
    example_theme/
      public/
        promoted-image.jpg
```

`promoted-image.jpg` should be a 420x220px JPEG image file. You could use this image file for example:



Then in `plugin.py`, register your public directory with CKAN by calling the `add_public_directory()` function. Add this line to the `update_config()` function:

```
def update_config(self, config):

    # Add this plugin's templates dir to CKAN's extra_template_paths, so
    # that CKAN will use this plugin's custom templates.
    toolkit.add_template_directory(config, 'templates')

    # Add this plugin's public dir to CKAN's extra_public_paths, so
    # that CKAN will use this plugin's custom static files.
    toolkit.add_public_directory(config, 'public')
```

If you now browse to `127.0.0.1:5000/promoted-image.jpg`, you should see your image file.

To replace the image on the front page with your custom image, we need to override the `promoted.html` template snippet. Create the following directory and file:

```
ckanext-example_theme/
  ckanext/
    example_theme/
      templates/
        home/
          snippets/
            promoted.html
```

Edit your new `promoted.html` snippet, and insert these contents:

```
{% ckan_extends %}

{% block home_image_caption %}
  {{ _("CKAN's data previewing tool has many powerful features") }}
{% endblock %}

{# Replace the promoted image. #}
{% block home_image_content %}
  <a class="media-image" href="#">
```

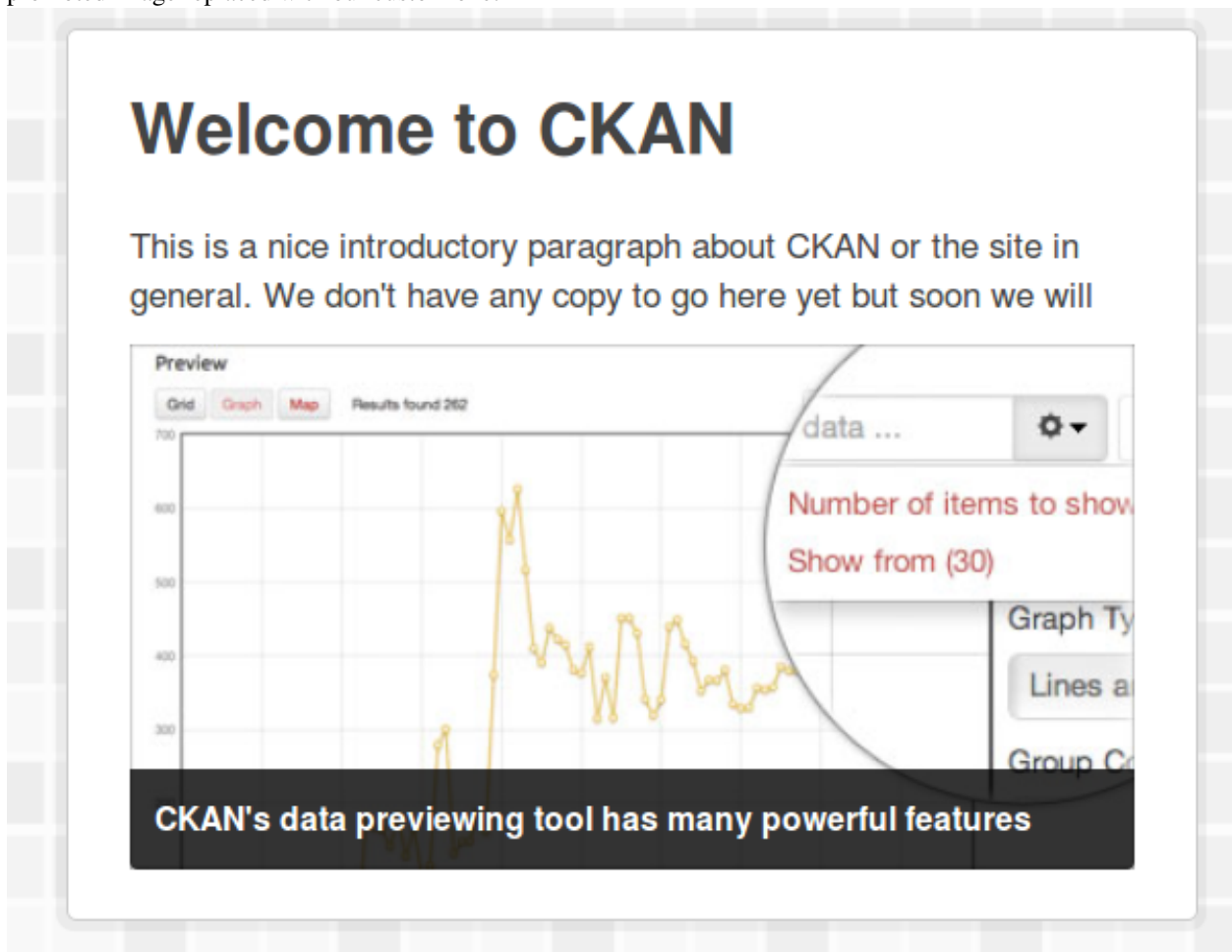
```

</a>
{% endblock %}
```

After calling `{% ckan_extends %}` to declare that it extends (rather than completely replaces) the default `promoted.html` snippet, this custom snippet overrides two of `promoted.html`'s template blocks. The first block replaces the caption text of the promoted image. The second block replaces the `` tag itself, pointing it at our custom static image file:

```
{% block home_image_content %}
<a class="media-image" href="#">
  
</a>
{% endblock %}
```

If you now restart the development web server and reload the [CKAN front page](#) in your browser, you should see the promoted image replaced with our custom one:



Customizing CKAN's CSS

See also:

There's nothing special about CSS in CKAN, once you've got started with editing CSS in CKAN (by following the tutorial below), then you just use the usual tools and techniques to explore and hack the CSS. We recommend using your browser's web development tools to explore and experiment with the CSS, then using any good text editor to edit your extension's CSS files as needed. For example:

Firefox developer tools These include a Page Inspector and a Style Editor

Firebug Another web development toolkit for Firefox

Chrome developer tools Tools for inspecting and editing CSS in Google Chrome

Mozilla Developer Network's CSS section A good collection of CSS documentation and tutorials

Extensions can add their own CSS files to modify or extend CKAN's default CSS. Create an `example_theme.css` file in your extension's `public` directory:

```
ckanext-example_theme/
  ckanext/
    example_theme/
      public/
        example_theme.css
```

Add this CSS into the `example_theme.css` file, to change the color of CKAN's "account masthead" (the bar across the top of the site that shows the logged-in user's account info):

```
.account-masthead {
  background-color: rgb(40, 40, 40);
}
```

If you restart the development web server you should be able to open this file at http://127.0.0.1:5000/example_theme.css in a web browser.

To make CKAN use our custom CSS we need to override the `base.html` template, this is the base template which the templates for all CKAN pages extend, so if we include a CSS file in this base template then the file will be included in every page of your CKAN site. Create the file:

```
ckanext-example_theme/
  ckanext/
    example_theme/
      templates/
        base.html
```

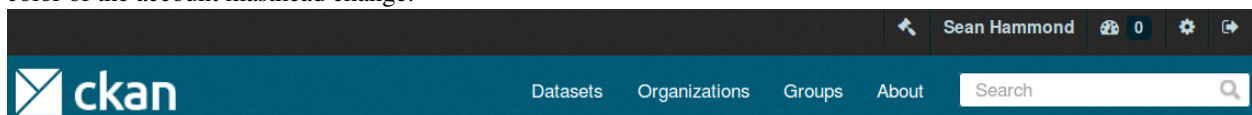
and put this Jinja code in it:

```
{% ckan_extends %}

{% block styles %}
  {{ super() }}
  <link rel="stylesheet" href="/example_theme.css" />
{% endblock %}
```

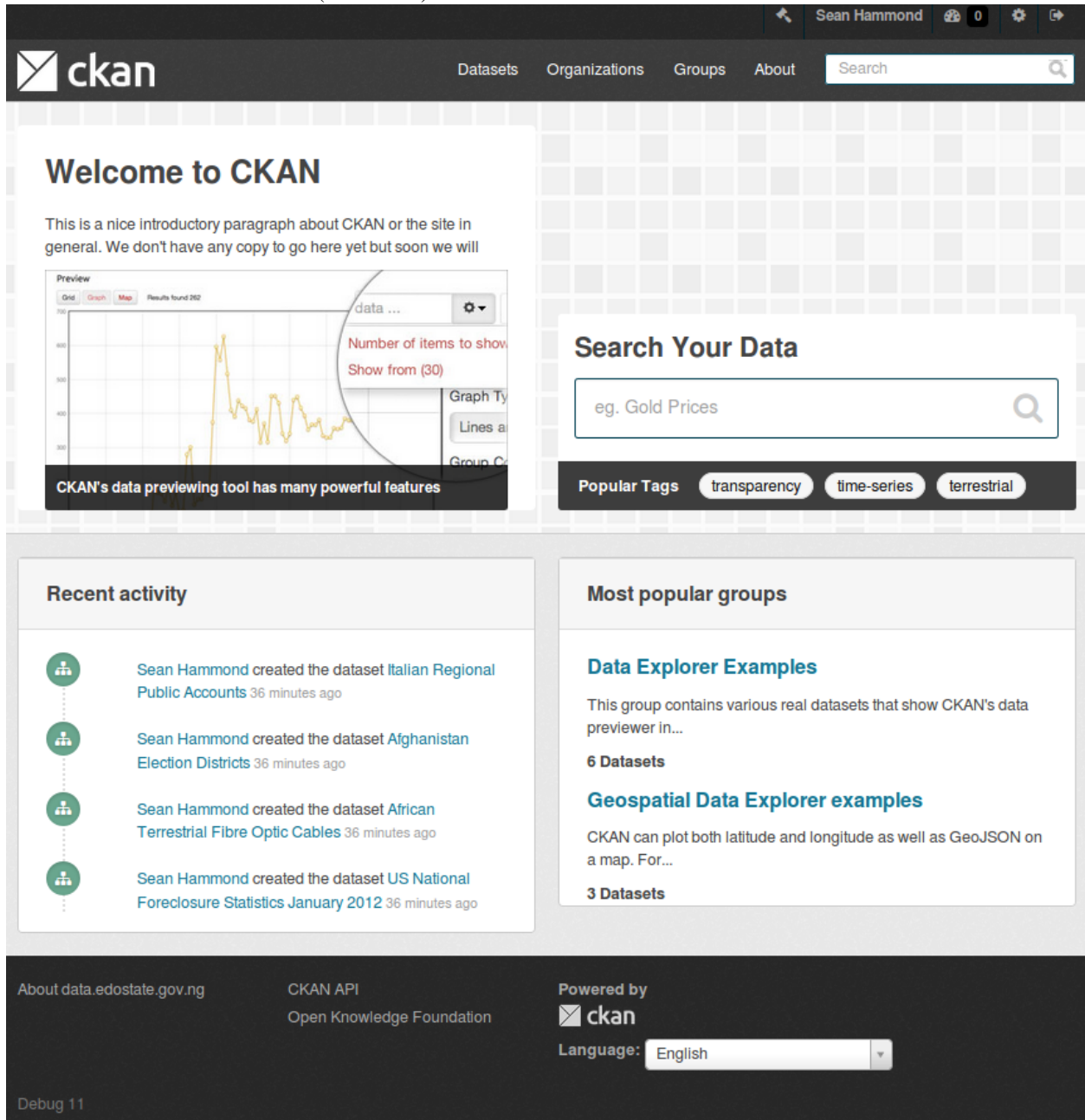
The default `base.html` template defines a `styles` block which can be extended to link to custom CSS files (any code in the `styles` block will appear in the `<head>` of the HTML page).

Restart the development web server and reload the CKAN page in your browser, and you should see the background color of the account masthead change:



This custom color should appear on all pages of your CKAN site.

Now that we have CKAN using our CSS file, we can add more CSS rules to the file and customize CKAN's CSS as much as we want. Let's add a bit more code to our `example_theme.css` file. This CSS implements a partial imitation of the [datahub.io](#) theme (circa 2013):



```
/* =====
The "account masthead" bar across the top of the site
===== */

.account-masthead {
    background-color: rgb(40, 40, 40);
}

/* The "bubble" containing the number of new notifications. */
.account-masthead .account .notifications a span {
    background-color: black;
}
```



```

}
/* The text and icons in the user account info. */
.account-masthead .account ul li a {
  color: rgba(255, 255, 255, 0.6);
}
/* The user account info text and icons, when the user's pointer is hovering
   over them. */
.account-masthead .account ul li a:hover {
  color: rgba(255, 255, 255, 0.7);
  background-color: black;
}

/* =====
   The main masthead bar that contains the site logo, nav links, and search
   ===== */

.masthead {
  background-color: #3d3d3d;
}
/* The "navigation pills" in the masthead (the links to Datasets,
   Organizations, etc) when the user's pointer hovers over them. */
.masthead .navigation .nav-pills li a:hover {
  background-color: rgb(48, 48, 48);
  color: white;
}
/* The "active" navigation pill (for example, when you're on the /dataset page
   the "Datasets" link is active). */
.masthead .navigation .nav-pills li.active a {
  background-color: rgb(74, 74, 74);
}
/* The "box shadow" effect that appears around the search box when it
   has the keyboard cursor's focus. */
.masthead input[type="text"]:focus {
  -webkit-box-shadow: inset 0px 0px 2px 0px rgba(0, 0, 0, 0.7);
  box-shadow: inset 0px 0px 2px 0px rgba(0, 0, 0, 0.7);
}

/* =====
   The content in the middle of the front page
   ===== */

/* Remove the "box shadow" effect around various boxes on the page. */
.box {
  box-shadow: none;
}
/* Remove the borders around the "Welcome to CKAN" and "Search Your Data"
   boxes. */
.hero .box {
  border: none;
}
/* Change the colors of the "Search Your Data" box. */
.homepage .module-search .module-content {
  color: rgb(68, 68, 68);
  background-color: white;
}
/* Change the background color of the "Popular Tags" box. */

```

```
.homepage .module-search .tags {
  background-color: rgb(61, 61, 61);
}
/* Remove some padding. This makes the bottom edges of the "Welcome to CKAN"
   and "Search Your Data" boxes line up. */
.module-content:last-child {
  padding-bottom: 0px;
}
.homepage .module-search {
  padding: 0px;
}
/* Add a border line between the top and bottom halves of the front page. */
.homepage [role="main"] {
  border-top: 1px solid rgb(204, 204, 204);
}

/* =====
   The footer at the bottom of the site
   ===== */

.site-footer,
body {
  background-color: rgb(40, 40, 40);
}
/* The text in the footer. */
.site-footer,
.site-footer label,
.site-footer small {
  color: rgba(255, 255, 255, 0.6);
}
/* The link texts in the footer. */
.site-footer a {
  color: rgba(255, 255, 255, 0.6);
}
```

Adding CSS and JavaScript files using Fanstatic

If you're adding CSS files to your theme, you can add them using [Fanstatic](#) rather than the simple *extra_public_paths* method described in [Adding static files](#). If you're adding a JavaScript module, you *must* use Fanstatic.

Using Fanstatic to add JavaScript and CSS files takes advantage of Fanstatic's features, such as automatically serving minified files in production, caching and bundling files together to reduce page load times, specifying dependencies between files so that the files a page needs (and only the files it needs) are always loaded, and other tricks to optimize page load times.

Note: CKAN will only serve *.js and *.css files as Fanstatic resources, other types of static files (eg. image files, PDF files) must be added using the *extra_public_paths* method described in [Adding static files](#).

Adding a custom JavaScript or CSS file to CKAN using Fanstatic is simple. We'll demonstrate by changing our previous custom CSS example (see [Customizing CKAN's CSS](#)) to serve the CSS file with Fanstatic.

1. First, create a `fanstatic` directory in your extension and move the CSS file from `public` into `fanstatic`:

```

ckanext-example_theme/
  ckanext/
    example_theme/
      public/
        promoted-image.jpg
      fanstatic/
        example_theme.css

```

2. Use CKAN's `add_resource()` function to register your fanstatic directory with CKAN. Edit the `update_config()` method in your `plugin.py` file:

```

def update_config(self, config):

    # Add this plugin's templates dir to CKAN's extra_template_paths, so
    # that CKAN will use this plugin's custom templates.
    toolkit.add_template_directory(config, 'templates')

    # Add this plugin's public dir to CKAN's extra_public_paths, so
    # that CKAN will use this plugin's custom static files.
    toolkit.add_public_directory(config, 'public')

    # Register this plugin's fanstatic directory with CKAN.
    # Here, 'fanstatic' is the path to the fanstatic directory
    # (relative to this plugin.py file), and 'example_theme' is the name
    # that we'll use to refer to this fanstatic directory from CKAN
    # templates.
    toolkit.add_resource('fanstatic', 'example_theme')

```

3. Finally, edit your extension's `templates/base.html` file and use CKAN's custom Jinja2 tag `{% resource %}` instead of the normal `<link>` tag to import the file:

```

{% ckan_extends %}

{% block styles %}
  {{ super() }}

  {# Import example_theme.css using Fanstatic.
    'example_theme/' is the name that the example_theme/fanstatic directory
    was registered with when the toolkit.add_resource() function was called.
    'example_theme.css' is the path to the CSS file, relative to the root of
    the fanstatic directory. #}
  {% resource 'example_theme/example_theme.css' %}
{% endblock %}

```

Note: You can put `{% resource %}` tags anywhere in any template, and Fanstatic will insert the necessary `<style>` and `<script>` tags to include your CSS and JavaScript files and their dependencies in the right places in the HTML output (CSS files in the HTML `<head>`, JavaScript files at the bottom of the page).

Resources will *not* be included on the line where the `{% resource %}` tag is.

Note: A config file can be used to configure how Fanstatic should serve each resource file (whether or not to bundle files, what order to include files in, whether to include files at the top or bottom of the page, dependencies between files, etc.) See [Resources](#) for details.

Customizing CKAN's JavaScript

JavaScript code in CKAN is broken down into *modules*: small, independent units of JavaScript code. CKAN themes can add JavaScript features by providing their own modules. This tutorial will explain the main concepts involved in CKAN JavaScript modules and walk you through the process of adding custom modules to themes.

See also:

This tutorial assumes a basic understanding of CKAN plugins and templating, see:

- [Extending guide](#)
- [Customizing CKAN's templates](#)

See also:

This tutorial assumes a basic understanding of JavaScript and jQuery, see:

- [JavaScript on the Mozilla Developer Network](#)
- [jQuery.com](#), including the [jQuery Learning Center](#)

See also:

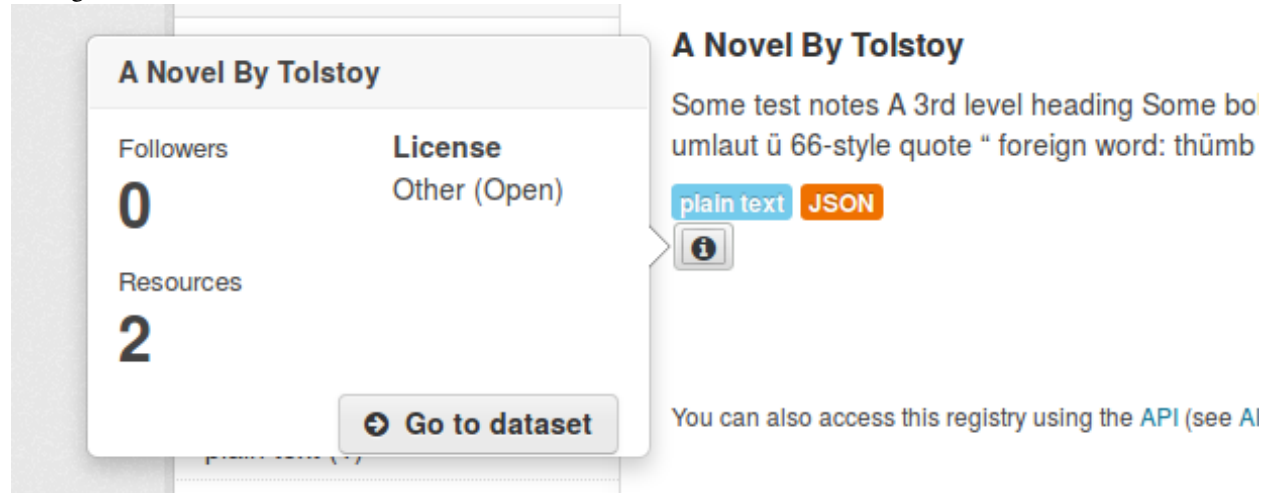
[String internationalization](#) How to mark strings for translation in your JavaScript code.

Overview

The idea behind CKAN's JavaScript modules is to keep the code simple and easy to test, debug and maintain, by breaking it down into small, independent modules. JavaScript modules in CKAN don't share global variables, and don't call each other's code.

These JavaScript modules are attached to HTML elements in the page, and enhance the functionality of those elements. The idea is that an HTML element with a JavaScript module attached should still be fully functional even if JavaScript is completely disabled (e.g. because the user's web browser doesn't support JavaScript). The user experience may not be quite as nice without JavaScript, but the functionality should still be there. This is a programming technique known as *graceful degradation*, and is a basic tenet of web accessibility.

In the sections below, we'll walk you through the steps to add a new JavaScript feature to CKAN - dataset info popovers. We'll add an info button to each dataset on the datasets page which, when clicked, opens a popover containing some extra information and user actions related to the dataset:



Initializing a JavaScript module

To get CKAN to call some custom JavaScript code, we need to:

1. Implement a JavaScript module, and register it with CKAN. Create the file `ckanext-example_theme/ckanext/example_theme/fanstatic/example_theme_popover.js`, with these contents:

```
// Enable JavaScript's strict mode. Strict mode catches some common
// programming errors and throws exceptions, prevents some unsafe actions from
// being taken, and disables some confusing and bad JavaScript features.
"use strict";

ckan.module('example_theme_popover', function ($, _) {
  return {
    initialize: function () {
      console.log("I've been initialized for element: ", this.el);
    }
  };
});
```

This bit of JavaScript calls the `ckan.module()` function to register a new JavaScript module with CKAN. `ckan.module()` takes two arguments: the name of the module being registered (`'example_theme_popover'` in this example) and a function that returns the module itself. The function takes two arguments, which we'll look at later. The module is just a JavaScript object with a single attribute, `initialize`, whose value is a function that CKAN will call to initialize the module. In this example, the `initialize` function just prints out a confirmation message - this JavaScript module doesn't do anything interesting yet.

Note: JavaScript module names should begin with the name of the extension, to avoid conflicting with other modules. See *JavaScript modules names should begin with the name of the extension*.

Note: Each JavaScript module's `initialize()` function is called on **DOM ready**.

2. Include the JavaScript module in a page, using Fanstatic, and apply it to one or more HTML elements on that page. We'll override CKAN's `package_item.html` template snippet to insert our module whenever a package is rendered as part of a list of packages (for example, on the dataset search page). Create the file `ckanext-example_theme/ckanext/example_theme/templates/snippets/package_item.html` with these contents:

```
{% ckan_extends %}

{% block content %}
  {{ super() }}

  {{# Use Fanstatic to include our custom JavaScript module.
  A <script> tag for the module will be inserted in the right place at the
  bottom of the page. #}}
  {% resource 'example_theme/example_theme_popover.js' %}

  {{# Apply our JavaScript module to an HTML element. The data-module attribute,
  which can be applied to any HTML element, tells CKAN to initialize an
  instance of the named JavaScript module for the element.
  The initialize() method of our module will be called with this HTML
  element as its this.el object. #}}
  <button data-module="example_theme_popover"
    class="btn"
```

```
        href="#">
        <i class="icon-info-sign"></i>
    </button>
{% endblock %}
```

See also:

Using `data-*` attributes on the Mozilla Developer Network.

If you now restart the development server and open <http://127.0.0.1:5000/dataset> in your web browser, you should see an extra info button next to each dataset shown. If you open a JavaScript console in your browser, you should see the message that your module has printed out.

See also:

Most web browsers come with built-in developer tools including a JavaScript console that lets you see text printed by JavaScript code to `console.log()`, a JavaScript debugger, and more. For example:

- [Firefox Developer Tools](#)
- [Firebug](#)
- [Chrome DevTools](#)

If you have more than one dataset on your page, you'll see the module's message printed once for each dataset. The `package_item.html` template snippet is rendered once for each dataset that's shown in the list, so your `<button>` element with the `data-module="example_theme_popover"` attribute is rendered once for each dataset, and CKAN creates a new instance of your JavaScript module for each of these `<button>` elements. If you view the source of your page, however, you'll see that `example_theme_popover.js` is only included with a `<script>` tag once. Fanstatic is smart enough to deduplicate resources.

Note: JavaScript modules *must* be included as Fanstatic resources, you can't add them to a public directory and include them using your own `<script>` tags.

`this.options` and `this.el`

Now let's start to make our JavaScript module do something useful: show a [Bootstrap popover](#) with some extra info about the dataset when the user clicks on the info button.

First, we need our Jinja template to pass some of the dataset's fields to our JavaScript module as *options*. Change `package_item.html` to look like this:

```
{% ckan_extends %}

{% block content %}
    {{ super() }}
    {% resource 'example_theme/example_theme_popover.js' %}

    {# Apply our JavaScript module to an HTML <button> element.
       The additional data-module-* attributes are options that will be passed
       to the JavaScript module. #}
    <button data-module="example_theme_popover"
            data-module-title="{{ package.title }}"
            data-module-license="{{ package.license_title }}"
            data-module-num_resources="{{ package.num_resources }}">
        <i class="icon-info-sign"></i>
    </button>
{% endblock %}
```

This adds some `data-module-*` attributes to our `<button>` element, e.g. `data-module-title="{{ package.title }}"` (`{{ package.title }}` is a *Jinja2 expression* that evaluates to the title of the dataset, CKAN passes the Jinja2 variable `package` to our template).

Warning: Although HTML 5 treats any attribute named `data-*` as a data attribute, only attributes named `data-module-*` will be passed as options to a CKAN JavaScript module. So we have to named our parameters `data-module-title` etc., not just `data-title`.

Now let's make use of these options in our JavaScript module. Change `example_theme_popover.js` to look like this:

```
"use strict";

/* example_theme_popover
 *
 * This JavaScript module adds a Bootstrap popover with some extra info about a
 * dataset to the HTML element that the module is applied to. Users can click
 * on the HTML element to show the popover.
 *
 * title - the title of the dataset
 * license - the title of the dataset's copyright license
 * num_resources - the number of resources that the dataset has.
 */
ckan.module('example_theme_popover', function ($, _) {
  return {
    initialize: function () {

      // Access some options passed to this JavaScript module by the calling
      // template.
      var num_resources = this.options.num_resources;
      var license = this.options.license;

      // Format a simple string with the number of resources and the license,
      // e.g. "3 resources, Open Data Commons Attribution License".
      var content = 'NUM resources, LICENSE'
        .replace('NUM', this.options.num_resources)
        .replace('LICENSE', this.options.license)

      // Add a Bootstrap popover to the HTML element (this.el) that this
      // JavaScript module was initialized on.
      this.el.popover({title: this.options.title,
        content: content,
        placement: 'left'});
    }
  };
});
```

Note: It's best practice to add a docstring to the top of a JavaScript module, as in the example above, briefly documenting what the module does and what options it takes. See *JavaScript modules should have docstrings*.

Any `data-module-*` attributes on the HTML element are passed into the JavaScript module in the object `this.options`:

```
var num_resources = this.options.num_resources;
var license = this.options.license;
```

A JavaScript module can access the HTML element that it was applied to through the `this.el` variable. To add a popover to our info button, we call Bootstrap's `popover()` function on the element, passing in an options object with some of the options that Bootstrap's popovers accept:

```
// Add a Bootstrap popover to the HTML element (this.el) that this
// JavaScript module was initialized on.
this.el.popover({title: this.options.title,
                 content: content,
                 placement: 'left'});
```

See also:

For other objects and functions available to JavaScript modules, see *Objects and methods available to JavaScript modules*.

Default values for options

Default values for JavaScript module options can be provided by adding an `options` object to the module. If the HTML element doesn't have a `data-module-*` attribute for an option, then the default will be used instead. For example...

Todo

Think of an example to do using default values.

Ajax, event handling and CKAN's JavaScript sandbox

So far, we've used simple JavaScript string formatting to put together the contents of our popover. If we want the popover to contain much more complex HTML we really need to render a template for it, using the full power of *Jinja2 templates* and CKAN's *template helper functions*. Let's edit our plugin to use a Jinja2 template to render the contents of the popups nicely.

First, edit `package_item.html` to make it pass a few more parameters to the JavaScript module using `data-module-*` attributes:

```
{% ckan_extends %}

{% block content %}
    {{ super() }}

    {% resource 'example_theme/example_theme_popover.js' %}
    {% resource 'example_theme/example_theme_popover.css' %}

    <button data-module="example_theme_popover"
           data-module-id="{{ package.id }}"
           data-module-title="{{ package.title }}"
           data-module-license_title="{{ package.license_title }}"
           data-module-num_resources="{{ package.num_resources }}"
           <i class="icon-info-sign"></i>
    </button>
{% endblock %}
```

We've also added a second `{% resource %}` tag to the snippet above, to include a custom CSS file. We'll see the contents of that CSS file later.

Next, we need to add a new template snippet to our extension that will be used to render the contents of the popovers. Create this `example_theme_popover.html` file:


```

ckanext-example_theme/
  ckanext/
    example_theme/
      templates/
        ajax_snippets/
          example_theme_popover.html

```

and put these contents in it:

```

{# The contents for a dataset popover.

   id - the id of the dataset
   num_resources - the dataset's number of resources
   license_title - the dataset's license title

#}
<div class="context-info">
  <div class="nums">
    <dl>

      <dt>{{ _('Followers') }}</dt>
      <dd>{{ h.get_action('dataset_follower_count', {'id': id}) }}</dd>

      <dt>{{ _('Resources') }}</dt>
      <dd>{{ num_resources }}</dd>

    </dl>
  </div>

  <div class="license">
    <dl>
      <dt>License</dt>
      <dd>{{ license_title }}</dd>
    </dl>
  </div>

  <div class="clearfix"></div>

  {{ h.follow_button('dataset', id) }}

  <a class="btn go-to-dataset"
    href="{{ h.url_for(controller='package', action='read', id=id) }}">
    <i class="icon-circle-arrow-right"></i>
    Go to dataset
  </a>

</div>

```

This is a Jinja2 template that renders some nice looking contents for a popover, containing a few bits of information about a dataset. It uses a number of CKAN's Jinja2 templating features, including marking user-visible strings for translation and calling template helper functions. See *Customizing CKAN's templates* for details about Jinja2 templating in CKAN.

Note: The new template file has to be in a `templates/ajax_snippets/` directory so that we can use the template from our JavaScript code using CKAN's `getTemplate()` function. Only templates from `ajax_snippets` directories are available from the `getTemplate()` function.

Next, edit `fanstatic/example_theme_popover.js` as shown below. There's a lot going on in this new

JavaScript code, including:

- Using [Bootstrap's popover API](#) to show and hide popovers, and set their contents.
- Using [jQuery's event handling API](#) to get our functions to be called when the user clicks on a button.
- Using a function from CKAN's [JavaScript sandbox](#).

The sandbox is a JavaScript object, available to all JavaScript modules as `this.sandbox`, that contains a collection of useful functions and variables.

`this.sandbox.client` is a CKAN API client written in JavaScript, that should be used whenever a JavaScript module needs to talk to the CKAN API, instead of modules doing their own HTTP requests.

`this.sandbox.client.getTemplate()` is a function that sends an asynchronous (ajax) HTTP request (i.e. send an HTTP request from JavaScript and receive the response in JavaScript, without causing the browser to reload the whole page) to CKAN asking for a template snippet to be rendered.

Hopefully the liberal commenting in the code below makes it clear enough what's going on:

```
"use strict";

kan.module('example_theme_popover', function ($, _) {
  return {
    initialize: function () {

      // proxyAll() ensures that whenever an _on*() function from this
      // JavaScript module is called, the `this` variable in the function will
      // be this JavaScript module object.
      //
      // You probably want to call proxyAll() like this in the initialize()
      // function of most modules.
      //
      // This is a shortcut function provided by CKAN, it wraps jQuery's
      // proxy() function: http://api.jquery.com/jQuery.proxy/
      $.proxyAll(this, /_on/);

      // Add a Bootstrap popover to the button. Since we don't have the HTML
      // from the snippet yet, we just set the content to "Loading..."
      this.el.popover({title: this.options.title, html: true,
        content: 'Loading...', placement: 'left'});

      // Add an event handler to the button, when the user clicks the button
      // our _onClick() function will be called.
      this.el.on('click', this._onClick);
    },

    // Whether or not the rendered snippet has already been received from CKAN.
    _snippetReceived: false,

    _onClick: function(event) {

      // Send an ajax request to CKAN to render the popover.html snippet.
      // We wrap this in an if statement because we only want to request
      // the snippet from CKAN once, not every time the button is clicked.
      if (!this._snippetReceived) {
        this.sandbox.client.getTemplate('example_theme_popover.html',
          this.options,
          this._onReceiveSnippet);

        this._snippetReceived = true;
      }
    }
  }
});
```

```

    },

    // CKAN calls this function when it has rendered the snippet, and passes
    // it the rendered HTML.
    _onReceiveSnippet: function(html) {

        // Replace the popover with a new one that has the rendered HTML from the
        // snippet as its contents.
        this.el.popover('destroy');
        this.el.popover({title: this.options.title, html: true,
                        content: html, placement: 'left'});
        this.el.popover('show');
    },

    });
});

```

Finally, we need some custom CSS to make the HTML from our new snippet look nice. In `package_item.html` above we added a `{% resource %}` tag to include a custom CSS file. Now we need to create that file, `ckanext-example_theme/ckanext/example_theme/fanstatic/example_theme_popover.css`:

```

.dataset-list .popover .nums {

    /* We're reusing the .nums class from the dataset read page,
     * but we don't want the border, margin and padding, get rid of them. */
    border: none;
    margin: 0;
    padding: 0;

    /* We want the license and numbers to appear side by side, so float the
     * numbers list to the left and make it take up just over half of
     * the width of the popover. */
    float: left;
    width: 55%;
}

.dataset-list .popover .license {

    /* Prevent the words in the license from being wrapped mid-word. */
    word-break: keep-all;
}

.dataset-list .popover .go-to-dataset {

    /* Float the "Go to dataset" button to the right side of the popover,
     * this puts some space between the two buttons. */
    float: right;
}

```

Restart CKAN, and your dataset popovers should be looking much better.

Error handling

What if our JavaScript makes an Ajax request to CKAN, such as our `getTemplate()` call above, and gets an error in response? We can simulate this by changing the name of the requested template file to one that doesn't exist:

```
this.sandbox.client.getTemplate('foobar.html',
                                this.options,
                                this._onReceiveSnippet);
```

If you reload the datasets page after making this change, you'll see that when you click on a popover its contents remain *Loading*.... If you have a development console open in your browser, you'll see the error response from CKAN each time you click to open a popover.

Our JavaScript module's `_onReceiveSnippet()` function is only called if the request gets a successful response from CKAN. `getTemplate()` also accepts a second callback function parameter that will be called when CKAN sends an error response. Add this parameter to the `getTemplate()` call:

```
    this.sandbox.client.getTemplate('foobar.html',
                                    this.options,
                                    this._onReceiveSnippet,
                                    this._onReceiveSnippetError);
  },
},
```

Now add the new error function to the JavaScript module:

```
_onReceiveSnippetError: function(error) {
  this.el.popover('destroy');

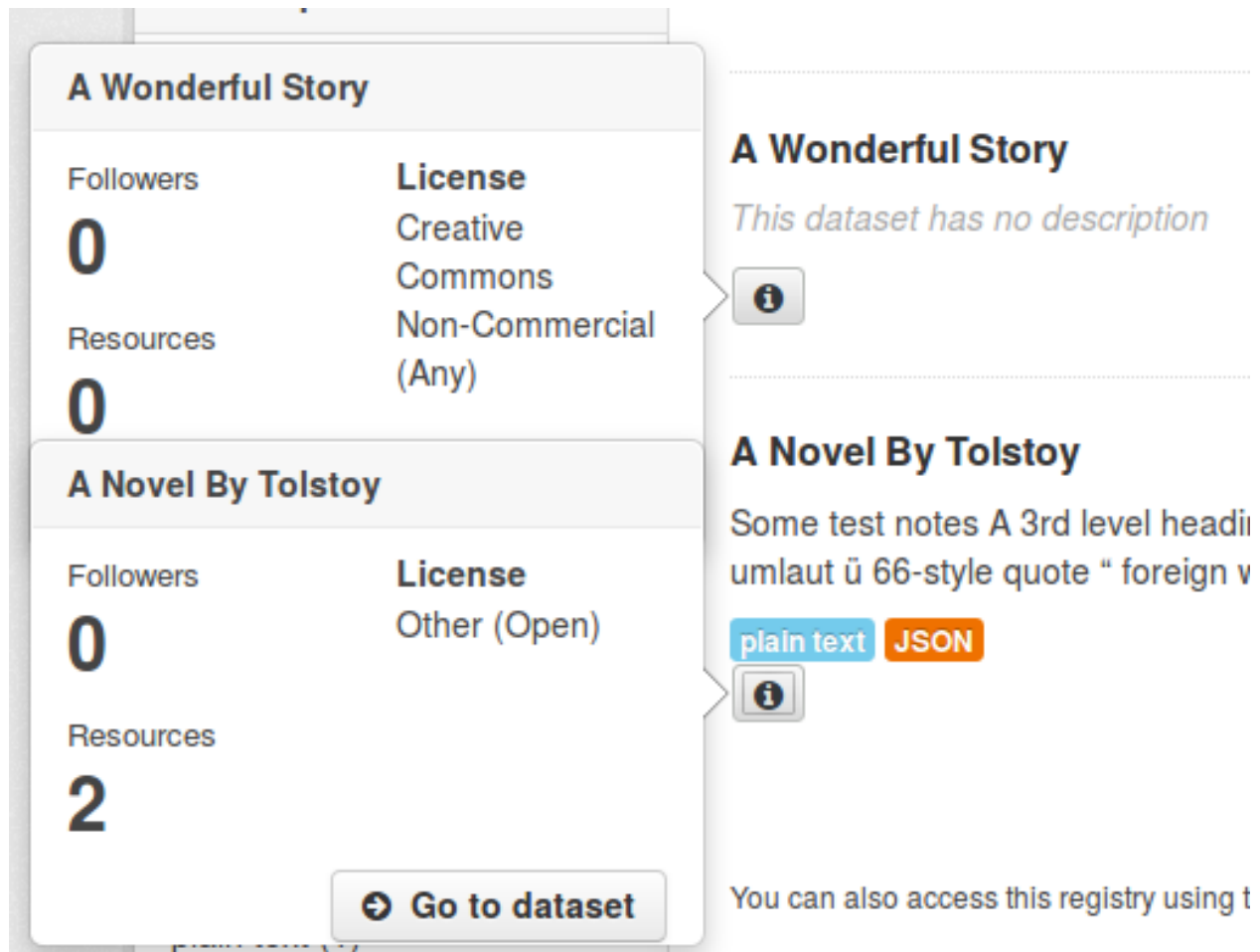
  var content = error.status + ' ' + error.statusText + ' :(';
  this.el.popover({title: this.options.title, html: true,
                  content: content, placement: 'left'});

  this.el.popover('show');
  this._snippetReceived = true;
},
```

After making these changes, you should see that if CKAN responds with an error, the contents of the popover are replaced with the error message from CKAN.

Pubsub

You may have noticed that, with our example code so far, if you click on the info button of one dataset on the page then click on the info button of another dataset, both dataset's popovers are shown. The first popover doesn't disappear when the second appears, and the popovers may overlap. If you click on all the info buttons on the page, popovers for all of them will be shown at once:



To make one popover disappear when another appears, we can use CKAN's `publish()` and `subscribe()` functions. These pair of functions allow different instances of a JavaScript module (or instances of different JavaScript modules) on the same page to talk to each other. The way it works is:

1. Modules can subscribe to events by calling `this.sandbox.client.subscribe()`, passing the 'topic' (a string that identifies the type of event to subscribe to) and a callback function.
2. Modules can call `this.sandbox.client.publish()` to publish an event for all subscribed modules to receive, passing the topic string and one or more further parameters that will be passed on as parameters to the receiver functions.
3. When a module calls `publish()`, any callback functions registered by previous calls to `subscribe()` with the same topic string will be called, and passed the parameters that were passed to `publish`.
4. If a module no longer wants to receive events for a topic, it calls `unsubscribe()`.

All modules that subscribe to events should have a `teardown()` function that unsubscribes from the event, to prevent memory leaks. CKAN calls the `teardown()` functions of modules when those modules are removed from the page. See *JavaScript modules should unsubscribe from events in `teardown()`*.

Warning: Don't tightly couple your JavaScript modules by overusing pubsub. See *Don't overuse pubsub*.

Remember that because we attach our `example_theme_popover.js` module to a `<button>` element that is rendered once for each dataset on the page, CKAN creates one instance of our module for each dataset. The only way these objects can communicate with each other so that one object can hide its popover when another object shows its

popover, is by using pubsub.

Here's a modified version of our `example_theme_popover.js` file that uses pubsub to make the dataset popovers disappear whenever a new popover appears:

```
"use strict";

ckan.module('example_theme_popover', function ($, _) {
  return {
    initialize: function () {
      $.proxyAll(this, /_on/);
      this.el.popover({title: this.options.title, html: true,
        content: 'Loading...', placement: 'left'});
      this.el.on('click', this._onClick);

      // Subscribe to 'dataset_popover_clicked' events.
      // Whenever any line of code publishes an event with this topic,
      // our _onPopoverClicked function will be called.
      this.sandbox.subscribe('dataset_popover_clicked',
        this._onPopoverClicked);
    },

    teardown: function() {
      this.sandbox.unsubscribe('dataset_popover_clicked',
        this._onPopoverClicked);
    },

    _snippetReceived: false,

    _onClick: function(event) {
      if (!this._snippetReceived) {
        this.sandbox.client.getTemplate('example_theme_popover.html',
          this.options,
          this._onReceiveSnippet);
        this._snippetReceived = true;
      }

      // Publish a 'dataset_popover_clicked' event for other interested
      // JavaScript modules to receive. Pass the button that was clicked as a
      // parameter to the receiver functions.
      this.sandbox.publish('dataset_popover_clicked', this.el);
    },

    // This callback function is called whenever a 'dataset_popover_clicked'
    // event is published.
    _onPopoverClicked: function(button) {

      // Wrap this in an if, because we don't want this object to respond to
      // its own 'dataset_popover_clicked' event.
      if (button !== this.el) {

        // Hide this button's popover.
        // (If the popover is not currently shown anyway, this does nothing).
        this.el.popover('hide');
      }
    },

    _onReceiveSnippet: function(html) {
      this.el.popover('destroy');
    }
  };
});
```

```

    this.el.popover({title: this.options.title, html: true,
                    content: html, placement: 'left'}));
    this.el.popover('show');
  },
};
});

```

jQuery plugins

CKAN provides a number of custom jQuery plugins for JavaScript modules to use by default, see [CKAN jQuery plugins reference](#). Extensions can also add their own jQuery plugins, and the plugins will then be available to all JavaScript code via the `this.$` object.

See also:

How to Create a Basic Plugin jQuery's own documentation on writing jQuery plugins. Read this for all the details on writing jQuery plugins, here we'll only provide a simple example and show you how to integrate it with CKAN.

It's a good idea to implement any JavaScript functionality not directly related to CKAN as a jQuery plugin. That way your CKAN JavaScript modules will be smaller as they'll contain only the CKAN-specific code, and your jQuery plugins will also be reusable on non-CKAN sites. CKAN core uses jQuery plugins to implement features including date formatting, warning users about unsaved changes when leaving a page containing a form without submitting the form, restricting the set of characters that can be typed into an input field, etc.

Let's add a jQuery plugin to our CKAN extension that makes our info buttons turn green when clicked.

Todo

Replace this with a more realistic example.

First we need to write the jQuery plugin itself. Create the file `ckanext-example_theme/ckanext/example_theme/fanstatic` with the following contents:

```

"use strict";

(function (jQuery) {

    jQuery.fn.greenify = function() {
        this.css( "color", "green" );
        return this;
    };

})(this.jQuery);

```

If this JavaScript code looks a little confusing at first, it's probably because it's using the [Immediately-Invoked Function Expression \(IIFE\)](#) pattern. This is a common JavaScript code pattern in which an anonymous function is created and then immediately called once, in a single expression. In the example above, we create an unnamed function that takes a single parameter, `jQuery`, and then we call the function passing `this.jQuery` to its `jQuery` parameter. The code inside the body of the function is the important part. Writing jQuery plugins in this way ensures that any variables defined inside the plugin are private to the plugin, and don't pollute the global namespace.

In the body of our jQuery plugin, we add a new function called `greenify()` to the `jQuery` object:

```

jQuery.fn.greenify = function() {
    this.css( "color", "green" );

```

```
    return this;
};
```

`jquery.fn` is the jQuery prototype object, the object that normal jQuery objects get all their methods from. By adding a method to this object, we enable any code that has a jQuery object to call our method on any HTML element or set of elements. For example, to turn all `<a>` elements on the page green you could do: `jQuery('a').greenify()`.

The code inside the `greenify()` function just calls jQuery's standard `css()` method to set the CSS `color` attribute of the element to `green`. This is just standard jQuery code, except that within a custom jQuery function you use `this` to refer to the jQuery object, instead of using `$` or `jquery` (as you would normally do when calling jQuery methods from code external to jQuery).

Our method then returns `this` to allow jQuery method chaining to be used with our method. For example, a user can set an element's CSS `color` attribute to `green` and add the CSS class `greenified` to the element in a single expression by chaining our jQuery method with another method: `$('#a').greenify().addClass('greenified');`

Before we can use our `greenify()` method in CKAN, we need to import the `jquery.greenify.js` file into the CKAN page. To do this, add a `{% resource %}` tag to a template file, just as you would do to include any other JavaScript or CSS file in CKAN. Edit the `package_item.html` file:

```
{% ckan_extends %}

{% block content %}
    {{ super() }}

    {% resource 'example_theme/example_theme_popover.js' %}
    {% resource 'example_theme/example_theme_popover.css' %}
    {% resource 'example_theme/jquery.greenify.js' %}

    <button data-module="example_theme_popover"
        data-module-id="{{ package.id }}"
        data-module-title="{{ package.title }}"
        data-module-license_title="{{ package.license_title }}"
        data-module-num_resources="{{ package.num_resources }}"
        <i class="icon-info-sign"></i>
    </button>
{% endblock %}
```

Now we can call the `greenify()` method from our `example_theme_popover` JavaScript module. For example, we could add a line to the `_onClick()` method in `example_theme_popover.js` so that when a dataset info button is clicked it turns green:

```
_onClick: function(event) {

    // Make all the links on the page turn green.
    this.$('i').greenify();

    if (!this._snippetReceived) {
        this.sandbox.client.getTemplate('example_theme_popover.html',
                                         this.options,
                                         this._onReceiveSnippet);

        this._snippetReceived = true;
    }
    this.sandbox.publish('dataset_popover_clicked', this.el);
},
```


Internationalization

Todo

Show how to Internationalize a JavaScript module.

Testing JavaScript modules

Todo

Show how to write tests for the example module.

Best practices for writing CKAN themes

Don't use `c`

As much as possible, avoid accessing the Pylons template context `c` (or `tmpl_context`). `c` is a thread-global variable, which encourages spaghetti code that's difficult to understand and to debug.

Instead, have controller methods add variables to the `extra_vars` parameter of `render()`, or have the templates call *template helper functions* instead.

`extra_vars` has the advantage that it allows templates, which are difficult to debug, to be simpler and shifts logic into the easier-to-test and easier-to-debug Python code. On the other hand, template helper functions are easier to reuse as they're available to all templates and they avoid inconsistencies between the namespaces of templates that are rendered by different controllers (e.g. one controller method passes the package dict as an extra var named `package`, another controller method passes the same thing but calls it `pkg`, a third calls it `pkg_dict`).

You can use the `ITemplateHelpers` plugin interface to add custom helper functions, see *Adding your own template helper functions*.

Use `url_for()`

Always use `url_for()` (available to templates as `h.url_for()`) when linking to other CKAN pages, instead of hardcoding URLs like ``. Links created with `url_for()` will update themselves if the URL routing changes in a new version of CKAN, or if a plugin changes the URL routing.

Use `{% trans %}`, `{% pluralize %}`, `_()` and `ungettext()`

All user-visible strings should be internationalized, see *String internationalization*.

Helper function names should begin with the name of the extension

Namespacing helper functions in this way avoids accidentally overriding, or being overridden by, a core helper function, or a helper function from another extension. For example:

```
def get_helpers(self):
    '''Register the most_popular_groups() function above as a template
    helper function.

    '''
    # Template helper function names should begin with the name of the
    # extension they belong to, to avoid clashing with functions from
    # other extensions.
    return {'example_theme_most_popular_groups': most_popular_groups}
```

Snippet filenames should begin with the name of the extension

Namespacing snippets in this way avoids accidentally overriding, or being overridden by, a core snippet, or a snippet from another extension. For example:

```
snippets/example_theme_most_popular_groups.html
```

JavaScript modules names should begin with the name of the extension

Namespacing JavaScript modules in this way avoids accidentally overriding, or being overridden by, a core module, or a module from another extension. For example: `fanstatic/example_theme_popover.js`:

```
// Enable JavaScript's strict mode. Strict mode catches some common
// programming errors and throws exceptions, prevents some unsafe actions from
// being taken, and disables some confusing and bad JavaScript features.
"use strict";

ckan.module('example_theme_popover', function ($, _) {
    return {
        initialize: function () {
            console.log("I've been initialized for element: ", this.el);
        }
    };
});
```

JavaScript modules should have docstrings

A JavaScript module should have a docstring at the top of the file, briefly documenting what the module does and what options it takes. For example:

```
"use strict";

/* example_theme_popover
 *
 * This JavaScript module adds a Bootstrap popover with some extra info about a
 * dataset to the HTML element that the module is applied to. Users can click
 * on the HTML element to show the popover.
 *
 * title - the title of the dataset
 * license - the title of the dataset's copyright license
 * num_resources - the number of resources that the dataset has.
 */
ckan.module('example_theme_popover', function ($, _) {
```

```

return {
  initialize: function () {

    // Access some options passed to this JavaScript module by the calling
    // template.
    var num_resources = this.options.num_resources;
    var license = this.options.license;

    // Format a simple string with the number of resources and the license,
    // e.g. "3 resources, Open Data Commons Attribution License".
    var content = 'NUM resources, LICENSE'
      .replace('NUM', this.options.num_resources)
      .replace('LICENSE', this.options.license)

    // Add a Bootstrap popover to the HTML element (this.el) that this
    // JavaScript module was initialized on.
    this.el.popover({title: this.options.title,
                     content: content,
                     placement: 'left'});

  }
};
});

```

JavaScript modules should unsubscribe from events in `teardown()`

Any JavaScript module that calls `this.sandbox.client.subscribe()` should have a `teardown()` function that calls `unsubscribe()`, to prevent memory leaks. CKAN calls the `teardown()` functions of modules when those modules are removed from the page.

Don't overuse pubsub

There shouldn't be very many cases where a JavaScript module really needs to use *Pubsub*, try to only use it when you really need to.

JavaScript modules in CKAN are designed to be small and loosely-coupled, for example modules don't share any global variables and don't call each other's functions. But pubsub offers a way to tightly couple JavaScript modules together, by making modules depend on multiple events published by other modules. This can make the code buggy and difficult to understand.

Use `{% snippet %}`, not `{% include %}`

Always use CKAN's custom `{% snippet %}` tag instead of Jinja's default `{% include %}` tag. Snippets can only access certain global variables, and any variables explicitly passed to them by the calling template. They don't have access to the full context of the calling template, as included files do. This makes snippets more reusable, and much easier to debug.

Snippets should have docstrings

A snippet should have a docstring comment at the top of the file that briefly documents what the snippet does and what parameters it requires. For example:

```
{#
Renders a list of the site's most popular groups.

groups - the list of groups to render

#}
<h3>Most popular groups</h3>
<ul>
  {% for group in groups %}
    <li>
      <a href="{{ h.url_for('group_read', action='read', id=group.name) }}">
        <h3>{{ group.display_name }}</h3>
      </a>
      {% if group.description %}
        <p>
          {{ h.markdown_extract(group.description, extract_length=80) }}
        </p>
      {% else %}
        <p>{{ _('This group has no description') }}</p>
      {% endif %}
      {% if group.packages %}
        <strong>{{ ungettext('{num} Dataset', '{num} Datasets', group.packages).format(num=group.packages) }}</strong>
      {% else %}
        <span>{{ _('0 Datasets') }}</span>
      {% endif %}
    </li>
  {% endfor %}
</ul>
```

Custom Jinja2 tags reference

Todo

Variables and functions available to templates

The following global variables and functions are available to all CKAN templates in their top-level namespace:

Note: In addition to the global variables listed below, each template also has access to variables from a few other sources:

- Any extra variables explicitly passed into a template by the controller that rendered the template will also be available to that template, in its top-level namespace. Any variables explicitly added to the template context variable `c` will also be available to the template as attributes of `c`.

To see which additional global variables and context attributes are available to a given template, use CKAN's *debug footer*.

- Any variables explicitly passed into a template snippet in the calling `{% snippet %}` tag will be available to the snippet in its top-level namespace. To see these variables, use the *debug footer*.
 - Jinja2 also makes a number of filters, tests and functions available in each template's global namespace. For a list of these, see the [Jinja2 docs](#).
-

tmpl_context

The [Pylons template context object](#), a thread-safe object that the application can store request-specific variables against without the variables associated with one HTTP request getting confused with variables from another request.

`tmpl_context` is usually abbreviated to `c` (an alias).

Using `c` in CKAN is discouraged, use template helper functions instead. See [Don't use c](#).

`c` is not available to snippets.

c

An alias for `tmpl_context`.

app_globals

The [Pylons App Globals object](#), an instance of the `ckan.lib.app_globals.Globals` class. The application can store request-independent variables against the `app_globals` object. Variables stored against `app_globals` are shared between all HTTP requests.

g

An alias for `app_globals`.

h

CKAN's [template helper functions](#), plus any *custom template helper functions* provided by any extensions.

request

The [Pylons Request object](#), contains information about the HTTP request that is currently being responded to, including the request headers and body, URL parameters, the requested URL, etc.

response

The [Pylons Response object](#), contains information about the HTTP response that is currently being prepared to be sent back to the user, including the HTTP status code, headers, cookies, etc.

session

The [Beaker session object](#), which contains information stored in the user's currently active session cookie.

_()

The `pylons.i18n.translation.gettext(value)` function:

Mark a string for translation. Returns the localized unicode string of value.

Mark a string to be localized as follows:

```
_( 'This should be in lots of languages' )
```

N_()

The `pylons.i18n.translation.gettext_noop(value)` function:

Mark a string for translation without translating it. Returns value.

Used for global strings, e.g.:

```
foo = N_('Hello')
```

```
class Bar:
```

```
    def __init__(self):
        self.local_foo = _(foo)
```

```
h.set_lang('fr')
```

```
assert Bar().local_foo == 'Bonjour'
```

```
h.set_lang('es')
```

```
assert Bar().local_foo == 'Hola'
```

```
assert foo == 'Hello'
```

ungettext ()

The `pylons.i18n.translation.ungettext(singular, plural, n)` function:

Mark a string for translation. Returns the localized unicode string of the pluralized value.

This does a plural-forms lookup of a message id. `singular` is used as the message id for purposes of lookup in the catalog, while `n` is used to determine which plural form to use. The returned message is a Unicode string.

Mark a string to be localized as follows:

```
ungettext('There is %(num)d file here', 'There are %(num)d files here',
n) % {'num': n}
```

translator

An instance of the `gettext.NullTranslations` class. This is for internal use only, templates shouldn't need to use this.

class actions

The `ckan.model.authz.Action` class.

Todo

Remove this? Doesn't appear to be used and doesn't look like something we want.

Objects and methods available to JavaScript modules

CKAN makes a few helpful objects and methods available for every JavaScript module to use, including:

- `this.el`, the HTML element that this instance of the object was initialized for. A jQuery element. See [this.options](#) and [this.el](#).
- `this.options`, an object containing any options that were passed to the module via `data-module-*` attributes in the template. See [this.options](#) and [this.el](#).
- `this.$()`, a jQuery find function that is scoped to the HTML element that the JavaScript module was applied to. For example, `this.$('a')` will return all of the `<a>` elements inside the module's HTML element, *not* all of the `<a>` elements on the entire page.

This is a shortcut for `this.el.find()`.

jQuery provides many useful features in an easy-to-use API, including document traversal and manipulation, event handling, and animation. See [jQuery's own docs](#) for details.

- `this.sandbox`, an object containing useful functions for all modules to use, including:
 - `this.sandbox.client`, an API client for calling the API
 - [Internationalization functions](#)
 - `this.sandbox.jquery`, a jQuery find function that is not bound to the module's HTML element. `this.sandbox.jquery('a')` will return all the `<a>` elements on the entire page. Using `this.sandbox.jquery` is discouraged, try to stick to `this.$` because it keeps JavaScript modules more independent.

See [JavaScript sandbox reference](#).

- A collection of [jQuery plugins](#).
- [Pubsub functions](#) that modules can use to communicate with each other, if they really need to.

- Bootstrap’s JavaScript features, see the [Bootstrap docs](#) for details.
- The standard JavaScript `window` object. Using `window` in CKAN JavaScript modules is discouraged, because it goes against the idea of a module being independent of global context. However, there are some circumstances where a module may need to use `window` (for example if a vendor plugin that the module uses needs it).
- `this.i18n`, a helper function for getting localized strings from `this.options`. English strings marked for translation can be added to the module’s `this.options` object, and `this.i18n()` can be called to retrieve the localized version of a string according to the current user’s language. See [Internationalization](#).
- `this.remove()`, a method that tears down the module and removes it from the page (this usually called by CKAN, not by the module itself).

Template helper functions reference

Helper functions

Consists of functions to typically be used within templates, but also available to Controllers. This module is available to templates as ‘h’.

`ckan.lib.helpers.redirect_to(*args, **kw)`

Issue a redirect: return an HTTP response with a 302 Moved header.

This is a wrapper for `routes.redirect_to()` that maintains the user’s selected language when redirecting.

The arguments to this function identify the route to redirect to, they’re the same arguments as `ckan.plugins.toolkit.url_for()` accepts, for example:

```
import ckan.plugins.toolkit as toolkit

# Redirect to /dataset/my_dataset.
toolkit.redirect_to(controller='package', action='read',
                    id='my_dataset')
```

Or, using a named route:

```
toolkit.redirect_to('dataset_read', id='changed')
```

`ckan.lib.helpers.url(*args, **kw)`

Create url adding i18n information if selected wrapper for pylons.url

`ckan.lib.helpers.get_site_protocol_and_host()`

Return the protocol and host of the configured `ckan.site_url`. This is needed to generate valid, full-qualified URLs.

If `ckan.site_url` is set like this:

```
ckan.site_url = http://example.com
```

Then this function would return a tuple (`'http'`, `'example.com'`) If the setting is missing, (`None`, `None`) is returned instead.

`ckan.lib.helpers.url_for(*args, **kw)`

Return the URL for the given controller, action, id, etc.

Usage:

```
import ckan.plugins.toolkit as toolkit

url = toolkit.url_for(controller='package', action='read',
```

```
        id='my_dataset')
=> returns '/dataset/my_dataset'
```

Or, using a named route:

```
toolkit.url_for('dataset_read', id='changed')
```

This is a wrapper for `routes.url_for()` that adds some extra features that CKAN needs.

```
ckan.lib.helpers.url_for_static(*args, **kw)
```

Returns the URL for static content that doesn't get translated (eg CSS)

It'll raise `CkanUrlException` if called with an external URL

This is a wrapper for `routes.url_for()`

```
ckan.lib.helpers.url_for_static_or_external(*args, **kw)
```

Returns the URL for static content that doesn't get translated (eg CSS), or external URLs

This is a wrapper for `routes.url_for()`

```
ckan.lib.helpers.is_url(*args, **kw)
```

Returns True if argument parses as a http, https or ftp URL

```
ckan.lib.helpers.url_is_local(url)
```

Returns True if url is local

```
ckan.lib.helpers.full_current_url()
```

Returns the fully qualified current url (eg <http://...>) useful for sharing etc

```
ckan.lib.helpers.current_url()
```

Returns current url unquoted

```
ckan.lib.helpers.lang()
```

Return the language code for the current locale eg *en*

```
ckan.lib.helpers.lang_native_name(lang=None)
```

Return the language name currently used in it's localised form either from parameter or current environ setting

```
class ckan.lib.helpers.Message(category, message, allow_html)
```

A message returned by `Flash.pop_messages()`.

Converting the message to a string returns the message text. Instances also have the following attributes:

- `message`: the message text.
- `category`: the category specified when the message was created.

```
ckan.lib.helpers.flash_notice(message, allow_html=False)
```

Show a flash message of type notice

```
ckan.lib.helpers.flash_error(message, allow_html=False)
```

Show a flash message of type error

```
ckan.lib.helpers.flash_success(message, allow_html=False)
```

Show a flash message of type success

```
ckan.lib.helpers.are_there_flash_messages()
```

Returns True if there are flash messages for the current user

```
ckan.lib.helpers.nav_link(text, *args, **kwargs)
```

Parameters

- **class** – pass extra class(es) to add to the `<a>` tag

- **icon** – name of ckan icon to use within the link
- **condition** – if `False` then no link is returned

`ckan.lib.helpers.build_nav_main(*args)`

build a set of menu items.

args: tuples of (menu type, title) eg ('login', _('Login')) outputs `title`

`ckan.lib.helpers.build_nav_icon(menu_item, title, **kw)`

Build a navigation item used for example in `user/read_base.html`.

Outputs `<i class="icon..."></i> title`.

Parameters

- **menu_item** (*string*) – the name of the defined menu item defined in config/routing as the named route of the same name
- **title** (*string*) – text used for the link
- **kw** – additional keywords needed for creating url eg `id=...`

Return type HTML literal

`ckan.lib.helpers.build_nav(menu_item, title, **kw)`

Build a navigation item used for example breadcrumbs.

Outputs `</i> title`.

Parameters

- **menu_item** (*string*) – the name of the defined menu item defined in config/routing as the named route of the same name
- **title** (*string*) – text used for the link
- **kw** – additional keywords needed for creating url eg `id=...`

Return type HTML literal

`ckan.lib.helpers.build_extra_admin_nav()`

Build extra navigation items used in `admin/base.html` for values defined in the config option `ckan.admin_tabs`. Typically this is populated by extensions.

Return type HTML literal

`ckan.lib.helpers.default_group_type()`

`ckan.lib.helpers.get_facet_items_dict(facet, limit=None, exclude_active=False)`

Return the list of unselected facet items for the given facet, sorted by count.

Returns the list of unselected facet constraints or facet items (e.g. tag names like “russian” or “tolstoy”) for the given search facet (e.g. “tags”), sorted by facet item count (i.e. the number of search results that match each facet item).

Reads the complete list of facet items for the given facet from `c.search_facets`, and filters out the facet items that the user has already selected.

Arguments: `facet` – the name of the facet to filter. `limit` – the max. number of facet items to return. `exclude_active` – only return unselected facets.

`ckan.lib.helpers.has_more_facets(facet, limit=None, exclude_active=False)`

Returns True if there are more facet items for the given facet than the limit.

Reads the complete list of facet items for the given facet from `c.search_facets`, and filters out the facet items that the user has already selected.

Arguments: facet – the name of the facet to filter. limit – the max. number of facet items. exclude_active – only return unselected facets.

`ckan.lib.helpers.unselected_facet_items (facet, limit=10)`

Return the list of unselected facet items for the given facet, sorted by count.

Returns the list of unselected facet constraints or facet items (e.g. tag names like “russian” or “tolstoy”) for the given search facet (e.g. “tags”), sorted by facet item count (i.e. the number of search results that match each facet item).

Reads the complete list of facet items for the given facet from `c.search_facets`, and filters out the facet items that the user has already selected.

Arguments: facet – the name of the facet to filter. limit – the max. number of facet items to return.

`ckan.lib.helpers.get_param_int (name, default=10)`

`ckan.lib.helpers.sorted_extras (package_extras, auto_clean=False, subs=None, exclude=None)`

Used for outputting package extras

Parameters

- **package_extras** (*dict*) – the package extras
- **auto_clean** (*bool*) – If true capitalize and replace `-_` with spaces
- **subs** (*dict* {*key*: *replacement*}) – substitutes to use instead of given keys
- **exclude** (*list of strings*) – keys to exclude

`ckan.lib.helpers.check_access (action, data_dict=None)`

`ckan.lib.helpers.linked_user (user, maxlength=0, avatar=20)`

`ckan.lib.helpers.group_name_to_title (name)`

`ckan.lib.helpers.markdown_extract (text, extract_length=190)`

return the plain text representation of markdown encoded text. That is the text without any html tags. If `extract_length` is 0 then it will not be truncated.

`ckan.lib.helpers.icon_url (name)`

`ckan.lib.helpers.icon_html (url, alt=None, inline=True)`

`ckan.lib.helpers.icon (name, alt=None, inline=True)`

`ckan.lib.helpers.resource_icon (res)`

`ckan.lib.helpers.format_icon (_format)`

`ckan.lib.helpers.dict_list_reduce (list_, key, unique=True)`

Take a list of dicts and create a new one containing just the values for the key with unique values if requested.

`ckan.lib.helpers.linked_gravatar (email_hash, size=100, default=None)`

`ckan.lib.helpers.gravatar (email_hash, size=100, default=None)`

`ckan.lib.helpers.pager_url (page, partial=None, **kwargs)`

class `ckan.lib.helpers.Page` (*collection*, *page=1*, *items_per_page=20*, *item_count=None*, *sqlalchemy_session=None*, *presliced_list=False*, *url=None*, ***kwargs*)

pager (**args*, ***kwargs*)

`ckan.lib.helpers.render_datetime (datetime_, date_format=None, with_hours=False)`

Render a datetime object or timestamp string as a localised date or in the requested format. If timestamp is badly formatted, then a blank string is returned.

Parameters

- **datetime** (*datetime or ISO string format*) – the date
- **date_format** (*string*) – a date format
- **with_hours** (*bool*) – should the *hours:mins* be shown

Return type string`ckan.lib.helpers.date_str_to_datetime(date_str)`

Convert ISO-like formatted datestring to datetime object.

This function converts ISO format date- and datetime-strings into datetime objects. Times may be specified down to the microsecond. UTC offset or timezone information may **not** be included in the string.

Note - Although originally documented as parsing ISO date(-times), this function doesn't fully adhere to the format. This function will throw a `ValueError` if the string contains UTC offset information. So in that sense, it is less liberal than ISO format. On the other hand, it is more liberal of the accepted delimiters between the values in the string. Also, it allows microsecond precision, despite that not being part of the ISO format.

`ckan.lib.helpers.parse_rfc_2822_date(date_str, assume_utc=True)`

Parse a date string of the form specified in RFC 2822, and return a datetime.

RFC 2822 is the date format used in HTTP headers. It should contain timezone information, but that cannot be relied upon.

If `date_str` doesn't contain timezone information, then the `'assume_utc'` flag determines whether we assume this string is local (with respect to the server running this code), or UTC. In practice, what this means is that if `assume_utc` is `True`, then the returned datetime is `'aware'`, with an associated `tzinfo` of offset zero. Otherwise, the returned datetime is `'naive'`.

If timezone information is available in `date_str`, then the returned datetime is `'aware'`, ie - it has an associated `tz_info` object.

Returns `None` if the string cannot be parsed as a valid datetime.

`ckan.lib.helpers.time_ago_from_timestamp(timestamp)`

Returns a string like *5 months ago* for a datetime relative to now :param timestamp: the timestamp or datetime :type timestamp: string or datetime

Return type string`ckan.lib.helpers.button_attr(enable, type='primary')``ckan.lib.helpers.dataset_display_name(package_or_package_dict)``ckan.lib.helpers.dataset_link(package_or_package_dict)``ckan.lib.helpers.resource_display_name(resource_dict)``ckan.lib.helpers.resource_link(resource_dict, package_id)``ckan.lib.helpers.related_item_link(related_item_dict)``ckan.lib.helpers.tag_link(tag)``ckan.lib.helpers.group_link(group)``ckan.lib.helpers.organization_link(organization)``ckan.lib.helpers.dump_json(obj, **kw)``ckan.lib.helpers.auto_log_message()``ckan.lib.helpers.activity_div(template, activity, actor, object=None, target=None)`

`ckan.lib.helpers.snippet(template_name, **kw)`

This function is used to load html snippets into pages. keywords can be used to pass parameters into the snippet rendering

`ckan.lib.helpers.convert_to_dict(object_type, objs)`

This is a helper function for converting lists of objects into lists of dicts. It is for backwards compatability only.

`ckan.lib.helpers.follow_button(obj_type, obj_id)`

Return a follow button for the given object type and id.

If the user is not logged in return an empty string instead.

Parameters

- **obj_type** (*string*) – the type of the object to be followed when the follow button is clicked, e.g. ‘user’ or ‘dataset’
- **obj_id** (*string*) – the id of the object to be followed when the follow button is clicked

Returns a follow button as an HTML snippet

Return type string

`ckan.lib.helpers.follow_count(obj_type, obj_id)`

Return the number of followers of an object.

Parameters

- **obj_type** (*string*) – the type of the object, e.g. ‘user’ or ‘dataset’
- **obj_id** (*string*) – the id of the object

Returns the number of followers of the object

Return type int

`ckan.lib.helpers.add_url_param(alternative_url=None, controller=None, action=None, extras=None, new_params=None)`

Adds extra parameters to existing ones

controller action & extras (dict) are used to create the base url via `url_for()` controller & action default to the current ones

This can be overridden providing an `alternative_url`, which will be used instead.

`ckan.lib.helpers.remove_url_param(key, value=None, replace=None, controller=None, action=None, extras=None, alternative_url=None)`

Remove one or multiple keys from the current parameters. The first parameter can be either a string with the name of the key to remove or a list of keys to remove. A specific key/value pair can be removed by passing a second value argument otherwise all pairs matching the key will be removed. If `replace` is given then a new param `key=replace` will be added. Note that the `value` and `replace` parameters only apply to the first key provided (or the only one provided if key is a string).

controller action & extras (dict) are used to create the base url via `url_for()` controller & action default to the current ones

This can be overridden providing an `alternative_url`, which will be used instead.

`ckan.lib.helpers.include_resource(resource)`

`ckan.lib.helpers.urls_for_resource(resource)`

Returns a list of urls for the resource specified. If the resource is a group or has dependencies then there can be multiple urls.

NOTE: This is for special situations only and is not the way to generally include resources. It is advised not to use this function.

`ckan.lib.helpers.debug_inspect (arg)`

Output pprint.pformat view of supplied arg

`ckan.lib.helpers.debug_full_info_as_list (debug_info)`

This dumps the template variables for debugging purposes only.

`ckan.lib.helpers.popular (type_, number, min=1, title=None)`

display a popular icon.

`ckan.lib.helpers.groups_available (am_member=False)`

Return a list of the groups that the user is authorized to edit.

Parameters `am_member` – if True return only the groups the logged-in user is a member of, otherwise return all groups that the user is authorized to edit (for example, sysadmin users are authorized to edit all groups) (optional, default: False)

`ckan.lib.helpers.organizations_available (permission='edit_group')`

Return a list of organizations that the current user has the specified permission for.

`ckan.lib.helpers.user_in_org_or_group (group_id)`

Check if user is in a group or organization

`ckan.lib.helpers.dashboard_activity_stream (user_id, filter_type=None, filter_id=None, offset=0)`

Return the dashboard activity stream of the current user.

Parameters

- `user_id (string)` – the id of the user
- `filter_type (string)` – the type of thing to filter by
- `filter_id (string)` – the id of item to filter by

Returns an activity stream as an HTML snippet

Return type string

`ckan.lib.helpers.recently_changed_packages_activity_stream (limit=None)`

`ckan.lib.helpers.escape_js (str_to_escape)`

Escapes special characters from a JS string.

Useful e.g. when you need to pass JSON to the templates

Parameters `str_to_escape` – string to be escaped

Return type string

`ckan.lib.helpers.get_pkg_dict_extra (pkg_dict, key, default=None)`

Returns the value for the dataset extra with the provided key.

If the key is not found, it returns a default value, which is None by default.

Parameters `pkg_dict` – dictized dataset

Key extra key to lookup

Default default value returned if not found

`ckan.lib.helpers.get_request_param (parameter_name, default=None)`

This function allows templates to access query string parameters from the request. This is useful for things like sort order in searches.

`ckan.lib.helpers.html_auto_link (data)`

Linkifies HTML

tag:... converted to a tag link dataset:... converted to a dataset link group:... converted to a group link [http://...](#)
converted to a link

`ckan.lib.helpers.render_markdown(data, auto_link=True, allow_html=False)`

Returns the data as rendered markdown

Parameters

- **auto_link** (*bool*) – Should ckan specific links be created e.g. *group:xxx*
- **allow_html** (*bool*) – If True then html entities in the markdown data. This is dangerous if users have added malicious content. If False all html tags are removed.

`ckan.lib.helpers.format_resource_items(items)`

Take a resource item list and format nicely with blacklisting etc.

`ckan.lib.helpers.resource_preview(resource, package)`

Returns a rendered snippet for a embedded resource preview.

Depending on the type, different previews are loaded. This could be an img tag where the image is loaded directly or an iframe that embeds a web page or a recline preview.

`ckan.lib.helpers.get_allowed_view_types(resource, package)`

`ckan.lib.helpers.rendered_resource_view(resource_view, resource, package, embed=False)`

Returns a rendered resource view snippet.

`ckan.lib.helpers.view_resource_url(resource_view, resource, package, **kw)`

Returns url for resource. made to be overridden by extensions. i.e by resource proxy.

`ckan.lib.helpers.resource_view_is_filterable(resource_view)`

Returns True if the given resource view support filters.

`ckan.lib.helpers.resource_view_get_fields(resource)`

Returns sorted list of text and time fields of a datastore resource.

`ckan.lib.helpers.resource_view_is_iframe(resource_view)`

Returns true if the given resource view should be displayed in an iframe.

`ckan.lib.helpers.resource_view_icon(resource_view)`

Returns the icon for a particular view type.

`ckan.lib.helpers.resource_view_display_preview(resource_view)`

Returns if the view should display a preview.

`ckan.lib.helpers.resource_view_full_page(resource_view)`

Returns if the edit view page should be full page.

`ckan.lib.helpers.remove_linebreaks(string)`

Remove linebreaks from string to make it usable in JavaScript

`ckan.lib.helpers.list_dict_filter(list_, search_field, output_field, value)`

Takes a list of dicts and returns the value of a given key if the item has a matching value for a supplied key

Parameters

- **list** (*list of dicts*) – the list to search through for matching items
- **search_field** (*string*) – the key to use to find matching items
- **output_field** (*string*) – the key to use to output the value
- **value** – the value to search for

`ckan.lib.helpers.SI_number_span(number)`

outputs a span with the number in SI unit eg 14700 -> 14.7k

```

ckan.lib.helpers.new_activities()
    Return the number of activities for the current user.

    See logic.action.get.dashboard_new_activities_count() for more details.

ckan.lib.helpers.uploads_enabled()

ckan.lib.helpers.get_featured_organizations(count=1)
    Returns a list of favourite organization in the form of organization_list action function

ckan.lib.helpers.get_featured_groups(count=1)
    Returns a list of favourite group the form of organization_list action function

ckan.lib.helpers.featured_group_org(items, get_action, list_action, count)

ckan.lib.helpers.get_site_statistics()

ckan.lib.helpers.resource_formats()
    Returns the resource formats as a dict, sourced from the resource format JSON file. key: potential user input value value: [canonical mimetype lowercased, canonical format (lowercase), human readable form] Fuller description of the fields are described in ckan/config/resource_formats.json.

ckan.lib.helpers.unified_resource_format(format)

ckan.lib.helpers.check_config_permission(permission)

ckan.lib.helpers.get_organization(org=None, include_datasets=False)

ckan.lib.helpers.license_options(existing_license_id=None)
    Returns [(l.title, l.id), ...] for the licenses configured to be offered. Always includes the existing_license_id, if supplied.

ckan.lib.helpers.mail_to(email_address, name)

ckan.lib.helpers.sanitize_id(id_)
    Given an id (uuid4), if it has any invalid characters it raises ValueError.

```

Template snippets reference

Todo

Autodoc all the default template snippets here. This probably means writing a Sphinx plugin.

JavaScript sandbox reference

Todo

Autodoc the JavaScript sandbox. This will probably require writing a custom Sphinx plugin.

JavaScript API client reference

Todo

Autodoc the JavaScript client. This will probably require writing a custom Sphinx plugin.

CKAN jQuery plugins reference

CKAN adds a number of custom plugins that can be accessed by JavaScript modules via `this.sandbox.jquery`.

Contributing guide

CKAN is free open source software and contributions are welcome, whether they're bug reports, source code, documentation or translations. The following sections will walk you through our processes for making different kinds of contributions to CKAN:

Reporting issues

If you've found a bug in CKAN, open a new issue on CKAN's [GitHub Issues](#) (try searching first to see if there's already an issue for your bug).

If you can fix the bug yourself, please *send a pull request*!

Todo

Could put more detail here about how to make a good bug report.

Translating CKAN

CKAN is used in many countries, and adding a new language to the web interface is a simple process.

CKAN uses the url to determine which language is used. An example would be `/fr/dataset` would be shown in french. If CKAN is running under a directory then an example would be `/root/fr/dataset`. For custom paths check the *ckan.root_path* config option.

See also:

Developers, see *String internationalization* for how to mark strings for translation in CKAN code.

Supported languages

CKAN already supports numerous languages. To check whether your language is supported, look in the source at `ckan/i18n` for translation files. Languages are named using two-letter ISO language codes (e.g. `es`, `de`).

If your language is present, you can switch the default language simply by setting the `ckan.locale_default` option in your CKAN config file, as described in *Internationalisation Settings*. For example, to switch to German:

```
ckan.locale_default=de
```

See also:

Internationalisation Settings

If your language is not supported yet, the remainder of this section provides instructions on how to prepare a translation file and add it to CKAN.

Adding a new language

If you want to add an entirely new language to CKAN, you have two options.

- *Transifex setup*. Creating translation files using Transifex, the open source translation software.
- *Manual setup*. Creating translation files manually in your own branch.

Note: Translating CKAN in Transifex is only enabled when a ‘call for translations’ is issued.

Transifex setup

Transifex, the open translation platform, provides a simple web interface for writing translations and is widely used for CKAN internationalization.

Using Transifex makes it easier to handle collaboration, with an online editor that makes the process more accessible.

Existing CKAN translation projects can be found at: <https://www.transifex.net/projects/p/ckan/teams/>

When leading up to a CKAN release, the strings are loaded onto Transifex and ckan-discuss list is emailed to encourage translation work. When the release is done, the latest translations on Transifex are checked back into CKAN.

Transifex administration

The Transifex workflow is as follows:

- Install transifex command-line utilities
- `tx init` in CKAN to connect to Transifex
- Run `python setup.py extract_messages` on the CKAN source
- Upload the local .pot file via command-line `tx push`
- Get people to complete translations on Transifex
- Pull locale .po files via `tx pull`
- `python setup.py compile_catalog`
- Git Commit and push po and mo files

Manual setup

Note: Please keep the CKAN core developers aware of new languages created in this way.

All the English strings in CKAN are extracted into the `ckan.pot` file, which can be found in `ckan/i18n`.

Note: For information, the pot file was created with the `babel` command `python setup.py extract_messages`.

0. Install Babel

You need Python's babel library (Debian package `python-pybabel`). Install it as follows with pip:

```
pip -E pyenv install babel
```

1. Create a 'po' file for your language

First, grab the CKAN i18n repository:

```
hg clone http://bitbucket.org/bboissin/ckan-i18n/
```

Then create a translation file for your language (a po file) using the pot file:

```
python setup.py init_catalog --locale YOUR_LANGUAGE
```

Replace `YOUR_LANGUAGE` with the two-letter ISO language code (e.g. `es`, `de`).

In future, when the pot file is updated, you can update the strings in your po file, while preserving your po edits, by doing:

```
python setup.py update_catalog --locale YOUR_LANGUAGE
```

2. Do the translation

Edit the po file and translate the strings. For more information on how to do this, see [the Pylons book](#).

We recommend using a translation tool, such as [poedit](#), to check the syntax is correct. There are also extensions for editors such as emacs.

3. Commit the translation

When the po is complete, create a branch in your source, then commit it to the CKAN i18n repo:

```
git checkout master
git branch translation-YOUR_LANGUAGE
git add ckan/i18n/YOUR_LANGUAGE/LC_MESSAGES/ckan.po
git commit -m '[i18n]: New language po added: YOUR_LANGUAGE' ckan/i18n/YOUR_LANGUAGE/LC_MESSAGES/ckan.po
git push origin translation-YOUR_LANGUAGE
```

4. Compile a translation

Once you have created a translation (either with Transifex or manually) you can build the po file into a mo file, ready for deployment.

With either method of creating the po file, it should be found in the CKAN i18n repository: `ckan/i18n/YOUR_LANGUAGE/LC_MESSAGES/ckan.po`

In this repo, compile the po file like this:

```
python setup.py compile_catalog --locale YOUR_LANGUAGE
```

As before, replace `YOUR_LANGUAGE` with your language short code, e.g. `es`, `de`.

This will result in a binary ‘mo’ file of your translation at `ckan/i18n/YOUR_LANGUAGE/LC_MESSAGES/ckan.mo`.

5. (optional) Deploy the translation

This section explains how to deploy your translation automatically to your host, if you are using a remote host.

It assumes a standard layout on the server (you may want to check before you upload!) and that you are deploying to `hu.ckan.net` for language `hu`.

Once you have a compiled translation file, for automated deployment to your host do:

```
fab config_0:hu.ckan.net upload_i18n:hu
```

See the `config_0` options if more configuration is needed e.g. of host or location.

Alternatively, if you do not want to use `fab`, you can just `scp`:

```
scp ckan.mo /home/okfn/var/srv/ckan.net/pyenv/src/ckan/ckan/i18n/hu/LC_MESSAGES/ckan.mo
```

6. Configure the language

Finally, once the `mo` file is in place, you can switch between the installed languages using the `ckan.locale` option in the CKAN config file, as described in [Internationalisation Settings](#).

Translations management policy

One of the aims of CKAN is to be accessible to the greatest number of users. Translating the user interface to as many languages as possible plays a huge part in this, and users are encouraged to contribute to the existing translations or submit a new one. At the same time we need to ensure the stability between CKAN releases, so the following guidelines apply when managing translations:

- Translations are open on Transifex as soon as a release branch for a minor version is created. At this point the strings are not yet frozen and can still change, but hopefully not too much. An announcement email will be sent to the mailing list and via Transifex.
- 2-3 weeks before the actual release the strings are frozen. This means that no new strings will be added on this release line. A second announce email is sent at this point.
- The translations will be kept open on Transifex after the release, and will be updated on each patch release, provided that:
 - They pass a review like any other change merged into a patch release (ie they must not introduce backwards incompatible changes).
 - Big changes like whole new languages (specially ones that introduce new features like RTL support, etc) should be tested and reviewed on a separate branch first.
 - Ultimately it is up to the committers and the release manager to decide if a new or updated translation is included in a patch release or needs to wait until the next minor release.
- The *master* branch is not currently translated. Translations from the latest stable line (see [CKAN releases](#)) are cherry-picked into master after each minor or patch release.

Testing CKAN

If you're a CKAN developer, if you're developing an extension for CKAN, or if you're just installing CKAN from source, you should make sure that CKAN's tests pass for your copy of CKAN. This section explains how to run CKAN's tests.

Install additional dependencies

Some additional dependencies are needed to run the tests. Make sure you've created a config file at `/etc/ckan/default/development.ini`, then activate your virtual environment:

```
. /usr/lib/ckan/default/bin/activate
```

Install nose and other test-specific CKAN dependencies into your virtual environment:

Changed in version 2.1: In CKAN 2.0 and earlier the requirements file was called `pip-requirements-test.txt`, not `dev-requirements.txt` as below.

```
pip install -r /usr/lib/ckan/default/src/ckan/dev-requirements.txt
```

Set up the test databases

Changed in version 2.1: Previously PostgreSQL tests used the databases defined in your `development.ini` file, instead of using their own test databases.

Create test databases:

```
sudo -u postgres createdb -O ckan_default ckan_test -E utf-8
sudo -u postgres createdb -O ckan_default datastore_test -E utf-8
paster datastore set-permissions -c test-core.ini | sudo -u postgres psql
```

This database connection is specified in the `test-core.ini` file by the `sqlalchemy.url` parameter.

Run the tests

To run CKAN's tests using PostgreSQL as the database, you have to give the `--with-pylons=test-core.ini` option on the command line. This command will run the tests for CKAN core and for the core extensions:

```
nosetests --ckan --with-pylons=test-core.ini ckan ckanext
```

The speed of the PostgreSQL tests can be improved by running PostgreSQL in memory and turning off durability, as described in the [PostgreSQL documentation](#).

By default the tests will keep the database between test runs. If you wish to drop and reinitialize the database before the run you can use the `reset-db` option:

```
nosetests --ckan --reset-db --with-pylons=test-core.ini ckan
```

Migration testing

If you're a CKAN developer or extension developer and your new code requires a change to CKAN's model, you'll need to write a migration script. To ensure that the migration script itself gets tested, you should run the tests with the `--ckan-migration` option, for example:

```
nosetests --ckan --ckan-migration --with-pylons=test-core.ini ckan ckanext
```

By default tests are run using the model defined in `ckan/model`. With the `--ckan-migration` option the tests will run using a database that has been created by running the migration scripts in `ckan/migration`, which is how the database is created and upgraded in production.

Warning: A common error when wanting to run tests against a particular database is to change `sqlalchemy.url` in `test.ini` or `test-core.ini`. The problem is that these are versioned files and people have checked in these by mistake, creating problems for other developers.

Common error messages

ConfigError

nose.config.ConfigError: Error reading config file 'setup.cfg': no such option 'with-pylons'

This error can result when you run `nosetests` for two reasons:

1. Pylons nose plugin failed to run. If this is the case, then within a couple of lines of running `nosetests` you'll see this warning: *Unable to load plugin pylons* followed by an error message. Fix the error here first'.
2. The Python module 'Pylons' is not installed into you Python environment. Confirm this with:

```
python -c "import pylons"
```

OperationalError

OperationalError: (OperationalError) no such function: plainto_tsquery ...

This error usually results from running a test which involves search functionality, which requires using a PostgreSQL database, but another (such as SQLite) is configured. The particular test is either missing a `@search_related` decorator or there is a mixup with the test configuration files leading to the wrong database being used.

nosetests

nosetests: error: no such option: --ckan Nose is either unable to find `ckan/ckan_nose_plugin.py` in the python environment it is running in, or there is an error loading it. If there is an error, this will surface it:

```
nosetests --version
```

There are a few things to try to remedy this:

Commonly this is because the `nosetests` isn't running in the python environment. You need to have nose actually installed in the python environment. To see which you are running, do this:

```
which nosetests
```

If you have activated the environment and this still reports `/usr/bin/nosetests` then you need to:

```
pip install --ignore-installed nose
```

If `nose --version` still fails, ensure that `ckan` is installed in your environment:

```
cd /usr/lib/ckan/default/src/ckan
python setup.py develop
```

One final check - the version of nose should be at least 1.0. Check with:

```
pip freeze | grep -i nose
```

Front-end Testing

All new CKAN features should be coded so that they work in the following browsers:

- Internet Explorer: 11, 10, 9 & 8
- Firefox: Latest + previous version
- Chrome: Latest + previous version

These browsers are determined by whatever has $\geq 1\%$ share with the latest months data from: <http://data.gov.uk/data/site-usage>

Install browser virtual machines

In order to test in all the needed browsers you'll need access to all the above browser versions. Firefox and Chrome should be easy whatever platform you are on. Internet Explorer is a little trickier. You'll need Virtual Machines.

We suggest you use <https://github.com/xdissent/ievms> to get your Internet Explorer virtual machines.

Testing methodology

Firstly we have a primer page. If you've touched any of the core front-end code you'll need to check if the primer is rendering correctly. The primer is located at: <http://localhost:5000/testing/primer>

Secondly whilst writing a new feature you should endeavour to test in at least in your core browser and an alternative browser as often as you can.

Thirdly you should fully test all new features that have a front-end element in all browsers before making your pull request into CKAN master.

Common pitfalls & their fixes

Here's a few of the most common front end bugs and a list of their fixes.

Reserved JS keywords

Since IE has a stricter language definition in JS it really doesn't like you using JS reserved keywords method names, variables, etc... This is a good list of keywords not to use in your JavaScript:

https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Reserved_Words

```
/* These are bad */
var a = {
  default: 1,
  delete: function() {}
};

/* These are good */
var a = {
  default_value: 1,
```

```
remove: function() {}  
};
```

Unclosed JS arrays / objects

Internet Explorer doesn't like it's JS to have unclosed JS objects and arrays. For example:

```
/* These are bad */  
var a = {  
  b: 'c',  
};  
var a = ['b', 'c', ];  
  
/* These are good */  
var a = {  
  c: 'c'  
};  
var a = ['b', 'c'];
```

Writing commit messages

We use the version control system [git](#) for our code and documentation, so when contributing code or docs you'll have to commit your changes to git and write a git commit message. Generally, follow the [commit guidelines from the Pro Git book](#):

- Try to make each commit a logically separate, digestible changeset.
- The first line of the commit message should concisely summarise the changeset.
- Optionally, follow with a blank line and then a more detailed explanation of the changeset.
- Use the imperative present tense as if you were giving commands to the codebase to change its behaviour, e.g. *Add tests for..., make xyzzy do frotz...*, this helps to make the commit message easy to read.

If your commit has an issue in the [CKAN issue tracker](#) put the issue number at the start of the first line of the commit message like this: `[#123]`. This makes the CKAN release manager's job much easier!

Here's an example of a good CKAN commit message:

```
[#607] Allow reactivating deleted datasets
```

```
Currently if a dataset is deleted and users navigate to the edit form,  
there is no state field and the delete button is still shown.
```

```
After this change, the state dropdown is shown if the dataset state is  
not active, and the delete button is not shown.
```

Making a pull request

Once you've written some CKAN code or documentation, you can submit it for review and merge into the central CKAN git repository by making a pull request. This section will walk you through the steps for making a pull request.

1. Create a git branch

Each logically separate piece of work (e.g. a new feature, a bug fix, a new docs page, or a set of improvements to a docs page) should be developed on its own branch forked from the master branch.

The name of the branch should include the issue number (if this work has an issue in the [CKAN issue tracker](#)), and a brief one-line synopsis of the work, for example:

```
2298-add-sort-by-controls-to-search-page
```

2. Fork CKAN on GitHub

Sign up for a free account on GitHub and [fork CKAN](#), so that you have somewhere to publish your work.

Add your CKAN fork to your local CKAN git repo as a git remote. Replace USERNAME with your GitHub username:

```
git remote add my_fork https://github.com/USERNAME/ckan
```

3. Commit and push your changes

Commit your changes on your feature branch, and push your branch to GitHub. For example, make sure you're currently on your feature branch then run these commands:

```
git add doc/my_new_feature.rst
git commit -m "Add docs for my new feature"
git push my_fork my_branch
```

When writing your git commit messages, try to follow the [Writing commit messages](#) guidelines.

4. Send a pull request

Once your work on a branch is complete and is ready to be merged into the master branch, [create a pull request on GitHub](#). A member of the CKAN team will review your work and provide feedback on the pull request page. The reviewer may ask you to make some changes. Once your pull request has passed the review, the reviewer will merge your code into the master branch and it will become part of CKAN!

When submitting a pull request:

- Your branch should contain one logically separate piece of work, and not any unrelated changes.
- You should have good commit messages, see [Writing commit messages](#).
- Your branch should contain new or changed tests for any new or changed code, and all the CKAN tests should pass on your branch, see [Testing CKAN](#).
- Your pull request shouldn't lower our test coverage. You can check it at our [coveralls page](https://coveralls.io/r/ckan/ckan) <<https://coveralls.io/r/ckan/ckan>>. If for some reason you can't avoid lowering it, explain why on the pull request.
- Your branch should contain new or updated documentation for any new or updated code, see [Writing documentation](#).
- Your branch should be up to date with the master branch of the central CKAN repo, so pull the central master branch into your feature branch before submitting your pull request.

For long-running feature branches, it's a good idea to pull master into the feature branch periodically so that the two branches don't diverge too much.

Reviewing and merging a pull request

Of course it's not possible to give an exact recipe for reviewing a pull request, you simply have to assess the code and decide whether you're happy with it. Nonetheless, here is an incomplete list of things to look for:

- Does the pull request contain one logically separate piece of work (e.g. one new feature, bug fix, etc. per pull request)?
- Does the pull request follow the guidelines for *writing commit messages*?
- Is the branch up to date - have the latest commits from master been pulled into the branch?
- Does the pull request contain new or updated tests for any new or updated code, and do the tests follow *CKAN's testing coding standards*?
- Do all the CKAN tests pass, on the new branch?
- Does the pull request contain new or updated docs for any new or updated features, and do the docs follow *CKAN's documentation guidelines*?
- Does the new code follow CKAN's code architecture and the various coding standards for Python, JavaScript, etc.?
- If the new code contains changes to the database schema, does it have a *database migration*?
- Does the code contain any changes that break backwards-incompatibility? If so, is the breakage necessary or do the benefits of the change justify the breakage? Have the breaking changes been added to the *changelog*?

Backwards-compatibility needs to be considered when making changes that break the interfaces that CKAN provides to third-party code, including API clients, plugins and themes.

In general, any code that's documented in the reference sections of the API, *extensions* or *theming* needs to be considered. For example this includes changes to the API actions, the plugin interfaces or plugins toolkit, the converter and validator functions (which are used by plugins), the custom Jinja2 tags and variables available to Jinja templates, the template helper functions, the core template files and their blocks, the sandbox available to JavaScript modules (including custom jQuery plugins and the JavaScript CKAN API client), etc.

- Does the new code add any dependencies to CKAN (e.g. new third-party Python modules imported)? If so, is the new dependency justified and has it been added following the right process? See *Upgrading CKAN's dependencies*.

Merging a pull request

Once you've reviewed a pull request and you're happy with it, you need to merge it into the master branch. You should do this using the `--no-ff` option in the `git merge` command. For example:

```
git checkout feature-branch
git pull origin feature-branch
git checkout master
git pull origin master
git merge --no-ff feature-branch
git push origin master
```

Before doing the `git push`, it's a good idea to check that all the tests are passing on your master branch (if the latest commits from master have already been pulled into the feature branch on github, then it may be enough to check that all tests passed for the latest commit on this branch on [Travis](#)).

Also before doing the `git push`, it's a good idea to use `git log` and/or `git diff` to check the difference between your local master branch and the remote master branch, to make sure you only push the changes you intend to push:

```
git log ...origin/master
git diff ..origin/master
```

Writing documentation

This section gives some guidelines to help us to write consistent and good quality documentation for CKAN.

Documentation isn't source code, and documentation standards don't need to be followed as rigidly as coding standards do. In the end, some documentation is better than no documentation, it can always be improved later. So the guidelines below are soft rules.

Having said that, we suggest just one hard rule: **no new feature (or change to an existing feature) should be missing from the docs** (but see [todo](#)).

See also:

Jacob Kaplan-Moss's [Writing Great Documentation](#) A series of blog posts about writing technical docs, a lot of our guidelines were based on this.

See also:

The quickest and easiest way to contribute documentation to CKAN is to sign up for a free GitHub account and simply edit the [CKAN Wiki](#). Docs started on the wiki can make it onto [docs.ckan.org](#) later. If you do want to contribute to [docs.ckan.org](#) directly, follow the instructions on this page.

Tip: Use the reStructuredText markup format when creating a wiki page, since reStructuredText is the format that docs.ckan.org uses, this will make moving the documentation from the wiki into docs.ckan.org later easier.

Getting started

This section will walk you through downloading the source files for CKAN's docs, editing them, and submitting your work to the CKAN project.

CKAN's documentation is created using [Sphinx](#), which in turn uses [Docutils](#) (reStructuredText is part of Docutils). Some useful links to bookmark:

- [Sphinx's reStructuredText Primer](#)
- [reStructuredText cheat sheet](#)
- [reStructuredText quick reference](#)
- [Sphinx Markup Constructs](#) is a full list of the markup that Sphinx adds on top of Docutils.

The source files for the docs are in [the doc directory of the CKAN git repo](#). The following sections will walk you through the process of making changes to these source files, and submitting your work to the CKAN project.

Install CKAN into a virtualenv

Create a [Python virtual environment](#) (virtualenv), activate it, install CKAN into the virtual environment, and install the dependencies necessary for building CKAN. In this example we'll create a virtualenv in a folder called `pyenv`. Run these commands in a terminal:

Changed in version 2.1: In CKAN 2.0 and earlier the requirements file was called `pip-requirements-docs.txt`, not `dev-requirements.txt` as below.

```
virtualenv --no-site-packages pyenv
. pyenv/bin/activate
pip install -e 'git+https://github.com/ckan/ckan.git#egg=ckan'
cd pyenv/src/ckan/
pip install -r dev-requirements.txt
pip install -r requirements.txt
```

Build the docs

You should now be able to build the CKAN documentation locally. Make sure your virtual environment is activated, and then run this command:

```
python setup.py build_sphinx
```

Now you can open the built HTML files in `build/sphinx/html`, e.g.:

```
firefox build/sphinx/html/index.html
```

Edit the reStructuredText files

To make changes to the documentation, use a text editor to edit the `.rst` files in `doc/`. Save your changes and then build the docs again (`python setup.py build_sphinx`) and open the HTML files in a web browser to preview your changes.

Once your docs are ready to submit to the CKAN project, follow the steps in [Making a pull request](#).

How the docs are organized

It's important that the docs have a clear, simple and extendable structure (and that we keep it that way as we add to them), so that both readers and writers can easily answer the questions: If you need to find the docs for a particular feature, where do you look? If you need to add a new page to the docs, where should it go?

As [Overview](#) explains, the documentation is organized into several guides, each for a different audience: a user guide for web interface users, an extending guide for extension developers, a contributing guide for core contributors, etc. These guides are ordered with the simplest guides first, and the most advanced last.

In the source, each one of these guides is a subdirectory with its own `index.rst` containing its own `.. toctree::` directive that lists all of the other files in that subdirectory. The root `toctree` just lists each of these `*/index.rst` files.

When adding a new page to the docs, the first question to ask yourself is: who is this page for? That should tell you which subdirectory to put your page in. You then need to add your page to that subdirectory's `index.rst` file.

Within each guide, the docs are broken up by topic. For example, the extending guide has a page for the writing extensions tutorial, a page about testing extensions, a page for the plugins toolkit reference, etc. Again, the topics are ordered with the simplest first and the most advanced last, and reference pages generally at the very end.

[The changelog](#) is one page that doesn't fit into any of the guides, because it's relevant to all of the different audiences and not only to one particular guide. So the changelog is simply a top-level page on its own. Hopefully we won't need to add many more of these top-level pages. If you're thinking about adding a page that serves two or more audiences at once, ask yourself whether you can break that up into separate pages and put each into one of the guides, then link them together using [seealso](#) boxes.

Within a particular page, for example a new page documenting a new feature, our suggestion for what sections the page might have is:

1. **Overview:** a conceptual overview of or introduction to the feature. Explain what the feature provides, why someone might want to use it, and introduce any key concepts users need to understand. This is the **why** of the feature.

If it's developer documentation (extension writing, theming, API, or core developer docs), maybe put an architecture guide [here](#).
2. **Tutorials:** tutorials and examples for how to setup the feature, and how to use the feature. This is the **how**.
3. **Reference:** any reference docs such as config options or API functions.

4. **Troubleshooting:** common error messages and problems, FAQs, how to diagnose problems.

Subdirectories

Some of the guides have subdirectories within them. For example *Maintainer's guide* contains a subdirectory *Installing CKAN* that collects together the various pages about installing CKAN with its own `doc/maintaining/installing/index.rst` file.

While subdirectories are useful, we recommend that you **don't put further subdirectories inside the subdirectories**, try to keep it to at most two levels of subdirectories inside the `doc` directory. Keep it simple, otherwise the structure becomes confusing, difficult to get an overview of and difficult to navigate.

Linear ordering

Keep in mind that Sphinx requires the docs to have a simple, linear ordering. With HTML pages it's possible to design structure where, for example, someone reads half of a page, then clicks on a link in the middle of the page to go and read another page, then goes back to the middle of the first page and continues reading where they left off. While technically you can do this in Sphinx as well, it isn't a good idea, things like the navigation links, table of contents, and PDF version will break, users will end up going in circles, and the structure becomes confusing.

So the pages of our Sphinx docs need to have a simple linear ordering - one page follows another, like in a book.

Sphinx

This section gives some useful tips about using Sphinx.

Don't introduce any new Sphinx warnings

When you build the docs, Sphinx prints out warnings about any broken cross-references, syntax errors, etc. We aim not to have any of these warnings, so when adding to or editing the docs make sure your changes don't introduce any new ones.

It's best to delete the `build` directory and completely rebuild the docs, to check for any warnings:

```
rm -rf build; python setup.py build_sphinx
```

Maximum line length

As with Python code, try to limit all lines to a maximum of 79 characters.

versionadded and versionchanged

Use Sphinx's `versionadded` and `versionchanged` directives to mark new or changed features. For example:

```
=====
Tag vocabularies
=====
```

```
.. versionadded:: 1.7
```

CKAN sites can have `*tag vocabularies*`, which are a way of grouping related tags together into custom fields.

...

With `versionchanged` you usually need to add a sentence explaining what changed (you can also do this with `versionadded` if you want):

```
=====
Authorization
=====
```

```
.. versionchanged:: 2.0
    Previous versions of CKAN used a different authorization system.
```

CKAN's authorization system controls which users are allowed to carry out which...

Cross-references and links

Whenever mentioning another page or section in the docs, an external website, a configuration setting, or a class, exception or function, etc. try to cross-reference it. Using proper Sphinx cross-references is better than just typing things like “see above/below” or “see section foo” because Sphinx cross-refs are hyperlinked, and because if the thing you’re referencing to gets moved or deleted Sphinx will update the cross-reference or print a warning.

Cross-referencing to another file

Use `:doc:` to cross-reference to other files by filename:

See `:doc:`configuration``

If the file you’re editing is in a subdir within the `doc` dir, you may need to use an absolute reference (starting with a `/`):

See `:doc:`/configuration``

See [Cross-referencing documents](#) for details.

Cross-referencing a section within a file

Use `:ref:` to cross-reference to particular sections within the same or another file. First you have to add a label before the section you want to cross-reference to:

```
.. _getting-started:
```

```
-----
Getting started
-----
```

then from elsewhere cross-reference to the section like this:

See `:ref:`getting-started``.

see [Cross-referencing arbitrary locations](#).

Cross-referencing to CKAN config settings

Whenever you mention a CKAN config setting, make it link to the docs for that setting in *Configuration Options* by using `:ref:` and the name of the config setting:

```
:ref:`ckan.site_title`
```

This works because all CKAN config settings are documented in *Configuration Options*, and every setting has a Sphinx label that is exactly the same as the name of the setting, for example:

```
.. _ckan.site_title:
```

```
ckan.site_title
^^^^^^^^^^^^^^^^
```

Example::

```
ckan.site_title = Open Data Scotland
```

```
Default value: ``CKAN``
```

This sets the name of the site, as displayed in the CKAN web interface.

If you add a new config setting to CKAN, make sure to document like this it in *Configuration Options*.

Cross-referencing to a Python object

Whenever you mention a Python function, method, object, class, exception, etc. cross-reference it using a Sphinx domain object cross-reference. See *Referencing other code objects with :py:*.

Changing the link text of a cross-reference

With `:doc:`, `:ref:` and other kinds of link, if you want the link text to be different from the title of the thing you're referencing, do this:

```
:doc:`the theming document </theming>`
```

```
:ref:`the getting started section <getting-started>`
```

Cross-referencing to an external page

The syntax for linking to external URLs is slightly different from cross-referencing, you have to add a trailing underscore:

```
`Link text <http://example.com/>`_
```

or to define a URL once and then link to it in multiple places, do:

```
This is `a link`_ and this is `a link`_ and this is  
`another link <a link>`_.
```

```
.. _a link: http://example.com/
```

see [Hyperlinks](#) for details.

Substitutions

Substitutions are a useful way to define a value that's needed in many places (eg. a command, the location of a file, etc.) in one place and then reuse it many times.

You define the value once like this:

```
.. |production.ini| replace:: /etc/ckan/default/production.ini
```

and then reuse it like this:

Now open your `|production.ini|` file.

`|production.ini|` will be replaced with the full value `/etc/ckan/default/production.ini`.

Substitutions can also be useful for achieving consistent spelling and capitalization of names like `reStructuredText`, `PostgreSQL`, `Nginx`, etc.

The `rst_epilog` setting in `doc/conf.py` contains a list of global substitutions that can be used from any file.

Substitutions can't immediately follow certain characters (with no space in-between) or the substitution won't work. If this is a problem, you can insert an escaped space, the space won't show up in the generated output and the substitution will work:

```
pip install -e 'git+\ |git_url|'
```

Similarly, certain characters are not allowed to immediately follow a substitution (without a space) or the substitution won't work. In this case you can just escape the following characters, the escaped character will show up in the output and the substitution will work:

```
pip install -e 'git+\ |git_url|\#egg=ckan'
```

Also see *Parsed literals* below for using substitutions in code blocks.

Parsed literals

Normally things like links and substitutions don't work within a literal code block. You can make them work by using a `parsed-literal` block, for example:

Copy your `development.ini` file to create a new `production.ini` file::

```
.. parsed-literal::

    cp |development.ini| |production.ini|
```

autodoc

We try to use `autodoc` to pull documentation from source code docstrings into our Sphinx docs, wherever appropriate. This helps to avoid duplicating documentation and also to keep the documentation closer to the code and therefore more likely to be kept up to date.

Whenever you're writing reference documentation for modules, classes, functions or methods, exceptions, attributes, etc. you should probably be using `autodoc`. For example, we use `autodoc` for the *Action API reference*, the *Plugin interfaces reference*, etc.

For how to write docstrings, see *Docstrings*.

todo

No new feature (or change to an existing feature) should be missing from the docs. It's best to document new features or changes as you implement them, but if you really need to merge something without docs then at least add a `todo directive` to mark where docs need to be added or updated (if it's a new feature, make a new page or section just to contain the `todo`):

```
=====
CKAN's builtin social network feature
=====

.. todo::

    Add docs for CKAN's builtin social network for data hackers.
```

deprecated

Use Sphinx's `deprecated directive` to mark things as deprecated in the docs:

```
.. deprecated:: 3.1
    Use :func:`spam` instead.
```

seealso

Often one page of the docs is related to other pages of the docs or to external pages. A `seealso block` is a nice way to include a list of related links:

```
.. seealso::

    :doc:`The DataStore extension <datastore>`
    A CKAN extension for storing data.

    CKAN's `demo site <http://demo.ckan.org/>`_
    A demo site running the latest CKAN beta version.
```

Seealso boxes are particularly useful when two pages are related, but don't belong next to each other in the same section of the docs. For example, we have docs about how to upgrade CKAN, these belong in the maintainer's guide because they're for maintainers. We also have docs about how to do a new release, these belong in the contributing guide because they're for developers. But both sections are about CKAN releases, so we link each to the other using seealso boxes.

Code examples

If you're going to paste example code into the docs, or add a tutorial about how to do something with code, then:

1. The code should be in standalone Python, HTML, JavaScript etc. files, not pasted directly into the `.rst` files. You then pull the code into your `.rst` file using a Sphinx `.. literalinclude::` directive (see example below).
2. The code in the standalone files should be a complete working example, with tests. Note that not all of the code from the example needs to appear in the docs, you can include just parts of it using `.. literalinclude::`, but the example code needs to be complete so it can be tested.

This is so that we don't end up with a lot of broken, outdated examples and tutorials in the docs because breaking changes have been made to CKAN since the docs were written. If your example code has tests, then when someone

makes a change in CKAN that breaks your example those tests will fail, and they'll know they have to fix their code or update your example.

The *plugins tutorial* is an example of this technique. `ckanext/example_iauthfunctions` is a complete and working example extension. The tests for the extension are in `ckanext/example_iauthfunctions/tests`. Different parts of the reStructuredText file for the tutorial pull in different parts of the example code as needed, like this:

```
.. literalinclude:: ../../ckanext/example_iauthfunctions/plugin_v3.py
   :start-after: # We have the logged-in user's user name, get their user id.
   :end-before: # Finally, we can test whether the user is a member of the curators group.
```

`literalinclude` has a few useful options for pulling out just the part of the code that you want. See the [Sphinx docs for literalinclude](#) for details.

You may notice that `ckanext/example_iauthfunctions` contains multiple versions of the same example plugin, `plugin_v1.py`, `plugin_v2.py`, etc. This is because the tutorial walks the user through first making a trivial plugin, and then adding more and more advanced features one by one. Each step of the tutorial needs to have its own complete, standalone example plugin with its own tests.

For more examples, look into the source files for other tutorials in the docs.

Style

This section covers things like what tone to use, how to capitalize section titles, etc. Having a consistent style will make the docs nice and easy to read and give them a complete, quality feel.

Use American spelling

Use American spellings everywhere: organization, authorization, realize, customize, initialize, color, etc. There's a list here: <https://wiki.ubuntu.com/EnglishTranslation/WordSubstitution>

Spellcheck

Please spellcheck documentation before merging it into master!

Commonly used terms

CKAN Should be written in ALL-CAPS.

email Use email not e-mail.

PostgreSQL, SQLAlchemy, Nginx, Python, SQLite, JavaScript, etc. These should always be capitalized as shown above (including capital first letters for Python and Nginx even when they're not the first word in a sentence). `doc/conf.py` defines substitutions for each of these so you don't have to remember them, see [Substitutions](#).

Web site Two words, with Web always capitalized

frontend Not front-end

command line Two words, not commandline or command-line (this is because we want to be like [Neal Stephenson](#))

CKAN config file or configuration file Not settings file, ini file, etc. Also, the **config file** contains **config options** such as `ckan.site_id`, and each config option is **set** to a certain **setting** or **value** such as `ckan.site_id = demo.ckan.org`.

Section titles

Capitalization in section titles should follow the same rules as in normal sentences: you capitalize the first word and any [proper nouns](#).

This seems like the easiest way to do consistent capitalization in section titles because it's a capitalization rule that we all know already (instead of inventing a new one just for section titles).

Right:

- Installing CKAN from package
- Getting started
- Command line interface
- Writing extensions
- Making an API request
- You're done!
- Libraries available to extensions

Wrong:

- Installing CKAN from Package
- Getting Started
- Command Line Interface
- Writing Extensions
- Making an API Request
- You're Done!
- Libraries Available To Extensions

For lots of examples of this done right, see [Django's table of contents](#).

In Sphinx, use the following section title styles:

```
=====
Top-level title
=====

-----
Second-level title
-----

Third-level title
=====

Fourth-level title
-----
```

If you need more than four levels of headings, you're probably doing something wrong, but see: <http://docutils.sourceforge.net/docs/ref/rst/restructuredtext.html#sections>

Be conversational

Write in a friendly, conversational and personal tone:

- Use contractions like don't, doesn't, it's etc.
- Use “we”, for example “*We'll publish a call for translations to the ckan-dev and ckan-discuss mailing lists, announcing that the new version is ready to be translated*” instead of “*A call for translations will be published*”.
- Refer to the reader personally as “you”, as if you're giving verbal instructions to someone in the room: “*First, you'll need to do X. Then, when you've done Y, you can start working on Z*” (instead of stuff like “*First X must be done, and then Y must be done...*”).

Write clearly and concretely, not vaguely and abstractly

Politics and the English Language has some good tips about this, including:

1. Never use a metaphor, simile, or other figure of speech which you are used to seeing in print.
2. Never use a long word where a short one will do.
3. If it's possible to cut out a word, always cut it out.
4. Never use the passive when you can be active.
5. Never use a foreign phrase, scientific word or jargon word if you can think of an everyday English equivalent.

This will make your meaning clearer and easier to understand, especially for people whose first language isn't English.

Facilitate skimming

Readers skim technical documentation trying to quickly find what's important or what they need, so break walls of text up into small, visually identifiable pieces:

- Use lots of [inline markup](#):

```
*italics*
**bold**
``code``
```

For code samples or filenames with variable parts, uses Sphinx's [:samp:](#) and [:file:](#) directives.

- Use [lists](#) to break up text.
- Use `.. note::` and `.. warning::`, see Sphinx's [paragraph-level markup](#).

(reStructuredText actually supports lots more of these: `attention`, `error`, `tip`, `important`, etc. but most Sphinx themes only style `note` and `warning`.)

- Break text into short paragraphs of 5-6 sentences each max.
- Use section and subsection headers to visualize the structure of a page.

Projects for beginner CKAN developers

The ‘[Good for Contribution](#)’ label on Github lists issues which are feasible for people who aren't intimately familiar with CKAN's internals. Many of them are things which would be extremely helpful if they got done, but the core team never seems to get around to them. They're all busy wrestling with the problems that do require familiarity with the internals. We hope this will make it easier for more people to assist the CKAN project, by giving new developers places to jump in.

These issues vary from simple text changes to more complicated code changes that require more knowledge of Python and CKAN. Do not despair if any individual task seems daunting; there's probably an easier one. If you have no programming skills, we can still use your help with documentation or translation.

If you wish to take up an issue make sure to keep in touch with the team on the Github issue, the mailing list or IRC.

CKAN code architecture

This section documents our CKAN-specific coding standards, which are guidelines for writing code that is consistent with the intended design and architecture of CKAN.

Encapsulate SQLAlchemy in `ckan.model`

Ideally SQLAlchemy should only be used within `ckan.model` and not from other packages such as `ckan.logic`. For example instead of using an SQLAlchemy query from the logic package to retrieve a particular user from the database, we add a `get()` method to `ckan.model.user.User`:

```
@classmethod
def get(cls, user_id):
    query = ...
    .
    .
    .
    return query.first()
```

Now we can call this method from the logic package.

Don't pass ORM objects to templates

Don't pass SQLAlchemy ORM objects (e.g. `ckan.model.User` objects) to templates (for example by adding them to `c`, passing them to `render()` in the `extra_vars` dict, returning them from template helper functions, etc.)

Using ORM objects in the templates often creates SQLAlchemy "detached instance" errors that cause 500 Server Errors and can be difficult to debug.

Instead, ORM objects should be dictized and their dictionary forms should be passed to templates. Controllers can dictize ORM objects using the functions in `ckan.lib.dictization`, but they should probably just get dictionaries from `ckan.logic.action` functions instead.

Always go through the action functions

Whenever some code, for example in `ckan.lib` or `ckan.controllers`, wants to get, create, update or delete an object from CKAN's model it should do so by calling a function from the `ckan.logic.action` package, and *not* by accessing `ckan.model` directly.

Action functions are exposed in the API

The functions in `ckan.logic.action` are exposed to the world as the [API guide](#). The API URL for an action function is automatically generated from the function name, for example `ckan.logic.action.create.package_create()` is exposed at `/api/action/package_create`. See [Steve Yegge's Google platforms rant](#) for some interesting discussion about APIs.

All publicly visible functions in the `ckan.logic.action.{create,delete,get,update}` namespaces will be exposed through the *API guide*. **This includes functions imported by those modules, as well as any helper functions** defined within those modules. To prevent inadvertent exposure of non-action functions through the action api, care should be taken to:

1. Import modules correctly (see *Imports*). For example:

```
import ckan.lib.search as search

search.query_for(...)
```

2. Hide any locally defined helper functions:

```
def _a_useful_helper_function(x, y, z):
    '''This function is not exposed because it is marked as private'''
    return x+y+z
```

3. Bring imported convenience functions into the module namespace as private members:

```
_get_or_bust = logic.get_or_bust
```

Use `get_action()`

Don't call `logic.action` functions directly, instead use `get_action()`. This allows plugins to override action functions using the `IActions` plugin interface. For example:

```
ckan.logic.get_action('group_activity_list_html')(...)
```

Instead of

```
ckan.logic.action.get.group_activity_list_html(...)
```

Auth functions and `check_access()`

Each action function defined in `ckan.logic.action` should use its own corresponding auth function defined in `ckan.logic.auth`. Instead of calling its auth function directly, an action function should go through `ckan.logic.check_access` (which is aliased `_check_access` in the action modules) because this allows plugins to override auth functions using the `IAuthFunctions` plugin interface. For example:

```
def package_show(context, data_dict):
    _check_access('package_show', context, data_dict)
```

`check_access` will raise an exception if the user is not authorized, which the action function should not catch. When this happens the user will be shown an authorization error in their browser (or will receive one in their response from the API).

`logic.get_or_bust()`

The `data_dict` parameter of logic action functions may be user provided, so required files may be invalid or absent. Naive Code like:

```
id = data_dict['id']
```

may raise a `KeyError` and cause CKAN to crash with a 500 Server Error and no message to explain what went wrong. Instead do:

```
id = _get_or_bust(data_dict, "id")
```

which will raise `ValidationError` if `"id"` is not in `data_dict`. The `ValidationError` will be caught and the user will get a 400 Bad Request response and an error message explaining the problem.

Validation and `ckan.logic.schema`

Logic action functions can use schema defined in `ckan.logic.schema` to validate the contents of the `data_dict` parameters that users pass to them.

An action function should first check for a custom schema provided in the context, and failing that should retrieve its default schema directly, and then call `_validate()` to validate and convert the data. For example, here is the validation code from the `user_create()` action function:

```
schema = context.get('schema') or ckan.logic.schema.default_user_schema()
session = context['session']
validated_data_dict, errors = _validate(data_dict, schema, context)
if errors:
    session.rollback()
    raise ValidationError(errors)
```

Controller & template helper functions

`ckan.lib.helpers` contains helper functions that can be used from `ckan.controllers` or from templates. When developing for ckan core, only use the helper functions found in `ckan.lib.helpers.__allowed_functions__`.

Deprecation

- Anything that may be used by *extensions*, *themes* or *API clients* needs to maintain backward compatibility at call-site. For example: action functions, template helper functions and functions defined in the plugins toolkit.
- The length of time of deprecation is evaluated on a function-by-function basis. At minimum, a function should be marked as deprecated during a point release.
- To deprecate a function use the `ckan.lib.maintain.deprecated()` decorator and add “deprecated” to the function’s docstring:

```
@maintain.deprecated("helpers.get_action() is deprecated and will be removed "
                    "in a future version of CKAN. Instead, please use the "
                    "extra_vars param to render() in your controller to pass "
                    "results from action functions to your templates.")
def get_action(action_name, data_dict=None):
    '''Calls an action function from a template. Deprecated in CKAN 2.3.'''
    if data_dict is None:
        data_dict = {}
    return logic.get_action(action_name)({}, data_dict)
```

- Any deprecated functions should be added to an *API changes and deprecations* section in the *Changelog* entry for the next release (do this before merging the deprecation into master)
- Keep the deprecation messages passed to the decorator short, they appear in logs. Put longer explanations of why something was deprecated in the changelog.

CSS coding standards

Note: For CKAN 2.0 we use LESS as a pre-processor for our core CSS. View [Front-end Documentation](#) for more information on this subject.

Formatting

All CSS documents must use **two spaces** for indentation and files should have no trailing whitespace. Other formatting rules:

- Use soft-tabs with a two space indent.
- Use double quotes.
- Use shorthand notation where possible.
- Put spaces after `:` in property declarations.
- Put spaces before `{` in rule declarations.
- Use hex color codes `#000` unless using `rgba()`.
- Always provide fallback properties for older browsers.
- Use one line per property declaration.
- Always follow a rule with one line of whitespace.
- Always quote `url()` and `@import()` contents.
- Do not indent blocks.

For example:

```
.media {
  overflow: hidden;
  color: #fff;
  background-color: #000; /* Fallback value */
  background-image: linear-gradient(black, grey);
}

.media .img {
  float: left;
  border: 1px solid #ccc;
}

.media .img img {
  display: block;
}

.media .content {
  background: #fff url("../images/media-background.png") no-repeat;
}
```

Naming

All ids, classes and attributes must be lowercase with hyphens used for separation.


```
/* GOOD */
.dataset-list {}

/* BAD */
.datasetlist {}
.datasetList {}
.dataset_list {}
```

Comments

Comments should be used liberally to explain anything that may be unclear at first glance, especially IE workarounds or hacks.

```
.prose p {
  font-size: 1.1666em /* 14px / 12px */;
}

.ie7 .search-form {
  /*
    Force the item to have layout in IE7 by setting display to block.
    See: http://reference.sitepoint.com/css/haslayout
  */
  display: inline-block;
}
```

Modularity and specificity

Try keep all selectors loosely grouped into modules where possible and avoid having too many selectors in one declaration to make them easy to override.

```
/* Avoid */
ul#dataset-list {}
ul#dataset-list li {}
ul#dataset-list li p a.download {}
```

Instead here we would create a dataset “module” and styling the item outside of the container allows you to use it on it’s own e.g. on a dataset page:

```
.dataset-list {}
.dataset-list-item {}
.dataset-list-item .download {}
```

In the same vein use classes make the styles more robust, especially where the HTML may change. For example when styling social links:

```
<ul class="social">
  <li><a href="">Twitter</a></li>
  <li><a href="">Facebook</a></li>
  <li><a href="">LinkedIn</a></li>
</ul>
```

You may use pseudo selectors to keep the HTML clean:

```
.social li:nth-child(1) a {
  background-image: url(twitter.png);
}
```

```
.social li:nth-child(2) a {
    background-image: url(facebook.png);
}

.social li:nth-child(3) a {
    background-image: url(linked-in.png);
}
```

However this will break any time the HTML changes for example if an item is added or removed. Instead we can use class names to ensure the icons always match the elements (Also you'd probably sprite the image :).

```
.social .twitter {
    background-image: url(twitter.png);
}

.social .facebook {
    background-image: url(facebook.png);
}

.social .linked-in {
    background-image: url(linked-in.png);
}
```

Avoid using tag names in selectors as this prevents re-use in other contexts.

```
/* Cannot use this class on an <ol> or <div> element */
ul.dataset-item {}
```

Also ids should not be used in selectors as it makes it far too difficult to override later in the cascade.

```
/* Cannot override this button style without including an id */
.btn#download {}
```

HTML coding standards

See also:

String internationalization How to mark strings for translation.

Formatting

All HTML documents must use **two spaces** for indentation and there should be no trailing whitespace. HTML5 syntax must be used and all attributes must use double quotes around attributes.

```
<video autoplay="autoplay" poster="poster_image.jpg">
  <source src="foo.ogg" type="video/ogg">
</video>
```

HTML5 elements should be used where appropriate reserving <div> and elements for situations where there is no semantic value (such as wrapping elements to provide styling hooks).

Doctype and layout

All documents must be using the HTML5 doctype and the <html> element should have a "lang" attribute. The <head> should also at a minimum include "viewport" and "charset" meta tags.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Example Site</title>
  </head>
  <body></body>
</html>
```

Forms

Form fields must always include a `<label>` element with a `"for"` attribute matching the `"id"` on the input. This helps accessibility by focusing the input when the label is clicked, it also helps screen readers match labels to their respective inputs.

```
<label for="field-email">email</label>
<input type="email" id="field-email" name="email" value="" />
```

Each `<input>` should have an `"id"` that is unique to the page. It does not have to match the `"name"` attribute.

Forms should take advantage of the new HTML5 input types where they make sense to do so, placeholder attributes should also be included where relevant. Including these can provide enhancements in browsers that support them such as tailored inputs and keyboards.

```
<div>
  <label for="field-email">Email</label>
  <input type="email" id="field-email" name="email" value="name@example.com">
</div>
<div>
  <label for="field-phone">Phone</label>
  <input type="phone" id="field-phone" name="phone" value="" placeholder="+44 077 12345 678">
</div>
<div>
  <label for="field-url">Homepage</label>
  <input type="url" id="field-url" name="url" value="" placeholder="http://example.com">
</div>
```

Wufoo provides an [excellent reference](#) for these attributes.

Including meta data

Classes should ideally only be used as styling hooks. If you need to include additional data in the HTML document, for example to pass data to JavaScript, then the HTML5 `data-` attributes should be used.

```
<a class="btn" data-format="csv">Download CSV</a>
```

These can then be accessed easily via jQuery using the `.data()` method.

```
jQuery('.btn').data('format'); //=> "csv"

// Get the contents of all data attributes.
jQuery('.btn').data(); => {format: "csv"}
```

One thing to note is that the JavaScript API for datasets will convert all attribute names into camelCase. So `"data-file-format"` will become `fileFormat`.

For example:

```
<a class="btn" data-file-format="csv">Download CSV</a>
```

Will become:

```
jQuery('.btn').data('fileFormat'); //=> "csv"  
jQuery('.btn').data(); => {fileFormat: "csv"}
```

Ideally you should be using CKAN's JavaScript module format for defining how JavaScript is initiated and interacts with the DOM.

Targeting Internet Explorer

Targeting lower versions of Internet Explorer (IE), those below version 9, should be handled by the stylesheets. Small fixes should be provided inline using the `.ie` specific class names. Larger fixes may require a separate stylesheet but try to avoid this if at all possible.

Adding IE specific classes:

```
<!DOCTYPE html>  
<!--[if lt IE 7]> <html lang="en" class="ie ie6"> <![endif]-->  
<!--[if IE 7]> <html lang="en" class="ie ie7"> <![endif]-->  
<!--[if IE 8]> <html lang="en" class="ie ie8"> <![endif]-->  
<!--[if gt IE 8]><!--> <html lang="en"> <!--<![endif]-->
```

Note: Only add lines for classes that are actually being used.

These can then be used within the CSS:

```
.clear:before,  
.clear:after {  
    content: "";  
    display: table;  
}  
  
.clear:after {  
    clear: both;  
}  
  
.ie7 .clear {  
    zoom: 1; /* For IE 6/7 (trigger hasLayout) */  
}
```

i18n

Don't include line breaks within `<p>` blocks. ie do this:

```
<p>Blah foo blah</p>  
<p>New paragraph, blah</p>
```

And **not**:

```
<p>Blah foo blah  
    New paragraph, blah</p>
```

JavaScript coding standards

See also:

String internationalization How to mark strings for translation.

Formatting

All JavaScript documents must use **two spaces** for indentation. This is contrary to the [OKFN Coding Standards](#) but matches what's in use in the current code base.

Coding style must follow the [idiomatic.js](#) style but with the following exceptions.

Note: Idiomatic is heavily based upon [Douglas Crockford's](#) style guide which is recommended by the [OKFN Coding Standards](#).

White space

Two spaces must be used for indentation at all times. Unlike in idiomatic whitespace must not be used `_inside_` parentheses between the parentheses and their Contents.

```
// BAD: Too much whitespace.
function getUrl( full ) {
  var url = '/styleguide/javascript/';
  if ( full ) {
    url = 'http://okfn.github.com/ckan' + url;
  }
  return url;
}

// GOOD:
function getUrl(full) {
  var url = '/styleguide/javascript/';
  if (full) {
    url = 'http://okfn.github.com/ckan' + url;
  }
  return url;
}
```

Note: See section 2.D.1.1 of idiomatic for more examples of this syntax.

Quotes

Single quotes should be used everywhere unless writing JSON or the string contains them. This makes it easier to create strings containing HTML.

```
jQuery('<div id="my-div" />').appendTo('body');
```

Object properties need not be quoted unless required by the interpreter.

```
var object = {
  name: 'bill',
  'class': 'user-name'
};
```

Variable declarations

One `var` statement must be used per variable assignment. These must be declared at the top of the function in which they are being used.

```
// GOOD:
var good = 'string';
var alsoGood = 'another';

// GOOD:
var good = 'string';
var okay = [
  'hmm', 'a bit', 'better'
];

// BAD:
var good = 'string',
    iffy = [
      'hmm', 'not', 'great'
    ];
```

Declare variables at the top of the function in which they are first used. This avoids issues with variable hoisting. If a variable is not assigned a value until later in the function then it is okay to define more than one per statement.

```
// BAD: contrived example.
function lowercaseNames(names) {
  var names = [];

  for (var index = 0, length = names.length; index < length; index += 1) {
    var name = names[index];
    names.push(name.toLowerCase());
  }

  var sorted = names.sort();
  return sorted;
}

// GOOD:
function lowercaseNames(names) {
  var names = [];
  var index, sorted, name;

  for (index = 0, length = names.length; index < length; index += 1) {
    name = names[index];
    names.push(name.toLowerCase());
  }

  sorted = names.sort();
  return sorted;
}
```

Naming

All properties, functions and methods must use lowercase camelCase:

```
var myUsername = 'bill';
var methods = {
  getSomething: function () {}
};
```

Constructor functions must use uppercase CamelCase:

```
function DatasetSearchView() {
}
```

Constants must be uppercase with spaces delimited by underscores:

```
var env = {
  PRODUCTION: 'production',
  DEVELOPMENT: 'development',
  TESTING: 'testing'
};
```

Event handlers and callback functions should be prefixed with “on”:

```
function onDownloadClick(event) {}

jQuery('.download').click(onDownloadClick);
```

Boolean variables or methods returning boolean functions should prefix the variable name with “is”:

```
function isAdmin() {}

var canEdit = isUser() && isAdmin();
```

Note: Alternatives are “has”, “can” and “should” if they make more sense

Private methods should be prefixed with an underscore:

```
View.extend({
  "click": "_onClick",
  _onClick: function (event) {
  }
});
```

Functions should be declared as named functions rather than assigning an anonymous function to a variable.

```
// GOOD:
function getName() {
}

// BAD:
var getName = function () {
};
```

Named functions are generally easier to debug as they appear named in the debugger.

Comments

Comments should be used to explain anything that may be unclear when you return to it in six months time. Single line comments should be used for all inline comments that do not form part of the documentation.

```
// Export the function to either the exports or global object depending
// on the current environment. This can be either an AMD module, CommonJS
// module or a browser.
if (typeof module.define === 'function' && module.define.amd) {
  module.define('broadcast', function () {
    return Broadcast;
  });
} else if (module.exports) {
  module.exports = Broadcast;
} else {
  module.Broadcast = Broadcast;
}
```

JSHint

All JavaScript should pass [JSHint](#) before being committed. This can be installed using npm (which is bundled with node) by running:

```
$ npm -g install jshint
```

Each project should include a jshint.json file with appropriate configuration options for the tool. Most text editors can also be configured to read from this file.

Documentation

For documentation we use a simple markup format to document all methods. The documentation should provide enough information to show the reader what the method does, arguments it accepts and a general example of usage. Also for API's and third party libraries, providing links to external documentation is encouraged.

The formatting is as follows:

```
/* My method description. Should describe what the method does and where
 * it should be used.
 *
 * param1 - The method params, one per line (default: null)
 * param2 - A default can be provided in brackets at the end.
 *
 * Example
 *
 * // Indented two spaces. Should give a common example of use.
 * client.getTemplate('index.html', {limit: 1}, function (html) {
 *   module.el.html(html);
 * });
 *
 * Returns describes what the object returns.
 */
```

For example:

```
/* Loads an HTML template from the CKAN snippet API endpoint. Template
 * variables can be passed through the API using the params object.
 *
 * Optional success and error callbacks can be provided or these can
 * be attached using the returns jQuery promise object.
 *
 * filename - The filename of the template to load.
```



```

* params    - An optional object containing key/value arguments to be
*             passed into the template.
* success   - An optional success callback to be called on load. This will
*             receive the HTML string as the first argument.
* error      - An optional error callback to be called if the request fails.
*
* Example
*
*   client.getTemplate('index.html', {limit: 1}, function (html) {
*     module.el.html(html);
*   });
*
* Returns a jqXHR promise object that can be used to attach callbacks.
*/

```

Testing

For unit testing we use the following libraries.

- **Mocha**: As a BDD unit testing framework.
- **Sinon**: Provides spies, stubs and mocks for methods and functions.
- **Chai**: Provides common assertions.

Tests are run from the `test/index.html` directory. We use the BDD interface (`describe()`, `it()` etc.) provided by mocha and the assert interface provided by chai.

Generally we try and have the core functionality of all libraries and modules unit tested.

Best practices

Forms

All forms should work without JavaScript enabled. This means that they must submit `application/x-www-form-urlencoded` data to the server and receive an appropriate response. The server should check for the `X-Requested-With: XMLHttpRequest` header to determine if the request is an ajax one. If so it can return an appropriate format, otherwise it should issue a 303 redirect.

The one exception to this rule is if a form or button is injected with JavaScript after the page has loaded. It's then not part of the HTML document and can submit any data format it pleases.

Ajax

Note: Calls to the CKAN API from JavaScript should be done through the CKAN client.

Ajax requests can be used to improve the experience of submitting forms and other actions that require server interactions. Nearly all requests will go through the following states.

1. User clicks button.
2. JavaScript intercepts the click and disables the button (add `disabled` attr).
3. A loading indicator is displayed (add class `.loading` to button).
4. The request is made to the server.

5. (a) On success the interface is updated.
(b) On error a message is displayed to the user if there is no other way to resolve the issue.
6. The loading indicator is removed.
7. The button is re-enabled.

Here's a possible example for submitting a search form using jQuery.

```
jQuery('#search-form').submit(function (event) {
    var form = $(this);
    var button = $('[type=submit]', form);

    // Prevent the browser submitting the form.
    event.preventDefault();

    button.prop('disabled', true).addClass('loading');

    jQuery.ajax({
        type: this.method,
        data: form.serialize(),
        success: function (results) {
            updatePageWithResults(results);
        },
        error: function () {
            showSearchError('Sorry we were unable to complete this search');
        },
        complete: function () {
            button.prop('disabled', false).removeClass('loading');
        }
    });
});
```

This covers possible issues that might arise from submitting the form as well as providing the user with adequate feedback that the page is doing something. Disabling the button prevents the form being submitted twice and the error feedback should hopefully offer a solution for the error that occurred.

Event handlers

When using event handlers to listen for browser events it's a common requirement to want to cancel the default browser action. This should be done by calling the `event.preventDefault()` method:

```
jQuery('button').click(function (event) {
    event.preventDefault();
});
```

It is also possible to return `false` from the callback function. Avoid doing this as it also calls the `event.stopPropagation()` method which prevents the event from bubbling up the DOM tree. This prevents other handlers listening for the same event. For example an analytics click handler attached to the `<body>` element.

Also jQuery (1.7+) now provides the `.on()` and `.off()` methods as alternatives to `.bind()`, `.unbind()`, `.delegate()` and `.undelegate()` and they should be preferred for all tasks.

Templating

Small templates that will not require customisation by the instance can be placed inline. If you need to create multi-line templates use an array rather than escaping newlines within a string:

```
var template = [
  '<li>',
  '<span></span>',
  '</li>'
].join('');
```

Always localise text strings within your templates. If you are including them inline this can always be done with jQuery:

```
jQuery(template).find('span').text(_('This is my text string'));
```

Larger templates can be loaded in using the CKAN snippet API. Modules get access to this functionality via the `sandbox.client` object:

```
initialize: function () {
  var el = this.el;
  this.sandbox.client.getTemplate('dataset.html', function (html) {
    el.html(html);
  });
}
```

The primary benefits of this is that the localisation can be done by the server and it keeps the JavaScript modules free from large strings.

Python coding standards

For Python code style follow [PEP 8](#) plus the guidelines below.

Some good links about Python code style:

- [Python Coding Standards](#) from Yahoo
- [Google Python Style Guide](#)

See also:

[String internationalization](#) How to mark strings for translation.

Use single quotes

Use single-quotes for string literals, e.g. `'my-identifier'`, *but* use double-quotes for strings that are likely to contain single-quote characters as part of the string itself (such as error messages, or any strings containing natural language), e.g. `"You've got an error!"`.

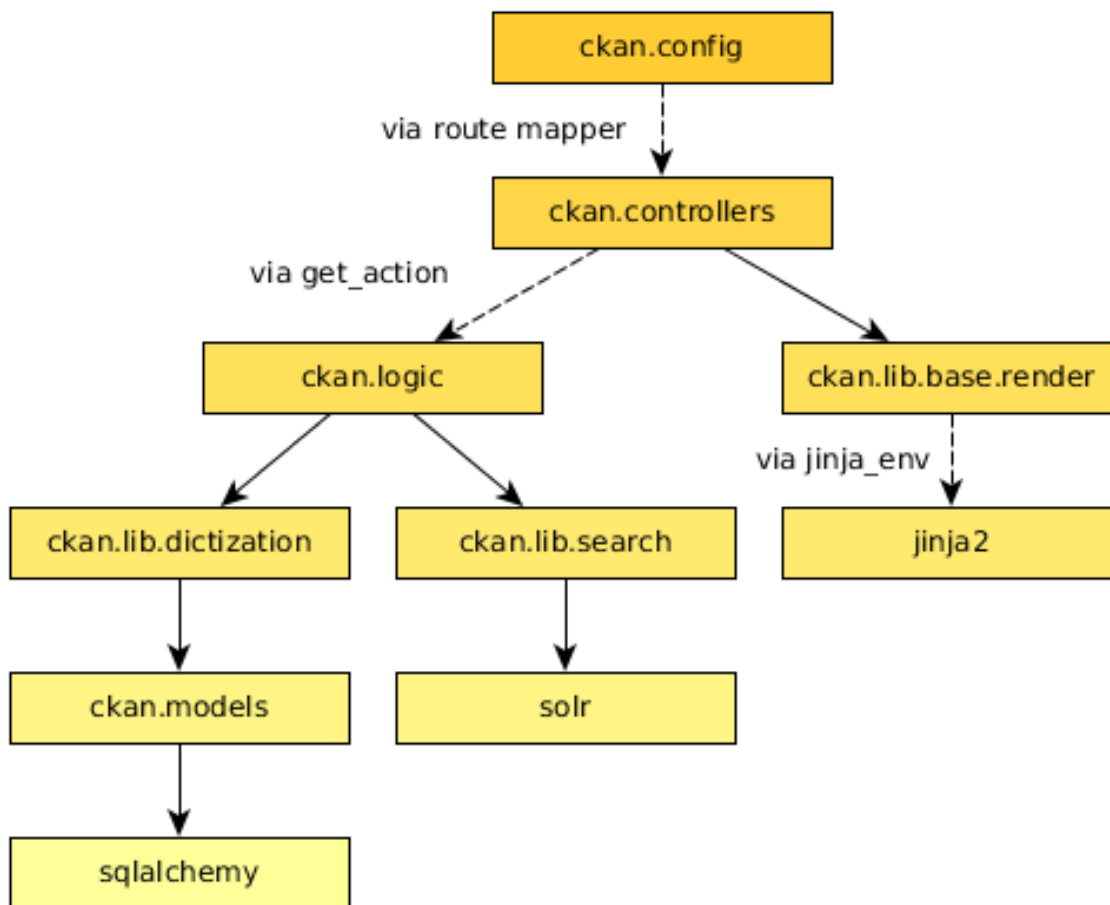
Single-quotes are easier to read and to type, but if a string contains single-quote characters then double-quotes are better than escaping the single-quote characters or wrapping the string in double single-quotes.

We also use triple single-quotes for docstrings, see [Docstrings](#).

Imports

- Avoid creating circular imports by only importing modules more specialized than the one you are editing.

CKAN often uses code imported into a data structure instead of importing names directly. For example CKAN controllers only use `get_action` to access logic functions. This allows customization by CKAN plugins.



- Don't use `from module import *`. Instead list the names you need explicitly:

```
from module import name1, name2
```

Use parenthesis around the names if they are longer than one line:

```
from module import (name1, name2, ...
                    name12, name13)
```

Most of the current CKAN code base imports just the modules and then accesses names with `module.name`. This allows circular imports in some cases and may still be necessary for existing code, but is not recommended for new code.

- Make all imports at the start of the file, after the module docstring. Imports should be grouped in the following order:
 1. Standard library imports
 2. Third-party imports
 3. CKAN imports

Logging

We use the Python standard library's logging module to log messages in CKAN, e.g.:

```
import logging
...
logger = logging.getLogger(__name__)
...
logger.debug('some debug message')
```

When logging:

- Keep log messages short.
- Don't include object representations in the log message. It *is* useful to include a domain model identifier where appropriate.
- Choose an appropriate log-level (DEBUG, INFO, ERROR, WARNING or CRITICAL, see [Python's Logging HOWTO](#)).

String formatting

Don't use the old %s style string formatting, e.g. "i am a %s" % sub. This kind of string formatting is not helpful for internationalization.

Use the new `.format()` method instead, and give meaningful names to each replacement field, for example:

```
_(' ... {foo} ... {bar} ...').format(foo='foo-value', bar='bar-value')
```

Docstrings

We want CKAN's docstrings to be clear and easy to read for programmers who are smart and competent but who may not know a lot of CKAN technical jargon and whose first language may not be English. We also want it to be easy to maintain the docstrings and keep them up to date with the actual behaviour of the code as it changes over time. So:

- All modules and all public functions, classes and methods exported by a module should normally have docstrings (see [PEP 257](#)).
- Keep docstrings short, describe only what's necessary and no more.
- Keep docstrings simple: use plain, concise English.
- Try to avoid repetition.

PEP 257 (Docstring Conventions)

Generally, follow [PEP 257](#) for docstrings. We'll only describe the ways that CKAN differs from or extends PEP 257 below.

CKAN docstrings deviate from PEP 257 in a couple of ways:

- We use `'''triple single quotes'''` around docstrings, not `"""triple double quotes"""` (put triple single quotes around one-line docstrings as well as multi-line ones, it makes them easier to expand later)
- We use Sphinx domain object cross-references to cross-reference to other code objects (see below)
- We use Sphinx directives for documenting parameters, exceptions and return values (see below)

Referencing other code objects with `:py:`

If you want to refer to another Python or JavaScript module, function or class etc. in a docstring (or from a `.rst` file), use [Sphinx domain object cross-references](#), for example:

```
See :py:mod:`ckan.lib.helpers`.
```

```
See :py:func:`ckan.logic.action.create.package_create`.
```

```
See :py:class:`ckan.logic.NotFound`.
```

For the full list of types of cross-reference, see the [Sphinx docs](#).

Note: These kinds of cross-references can also be used to reference other types of object besides Python objects, for example [JavaScript objects](#) or even command-line scripts and options and environment variables. See [the Sphinx docs](#) for the full details.

Cross-referencing objects like this means that Sphinx will style the reference with the right CSS, and hyperlink the reference to the docs for the referenced object. Sphinx can also generate error messages when non-existent objects are referenced, which helps to keep the docs up to date as the code changes.

Tip: Sphinx will render a cross-reference like `:py:func:`ckan.logic.action.create.package_create`` as the full name of the function: `ckan.logic.action.create.package_create()`. If you want the docs to contain only the local name of the function (e.g. just `package_create()`), put a `~` at the start:

```
:py:func:`~ckan.logic.action.create.package_create`
```

(But you should always use the fully qualified name in your docstring or `*.rst` file.)

Documenting exceptions raised with `:raises`

There are a few guidelines that CKAN code should follow regarding exceptions:

1. **All public functions that CKAN exports for third-party code to use should document any exceptions they raise.** See below for how to document exceptions raised.

For example the template helper functions in `ckan.lib.helpers`, anything imported into `ckan.plugins.toolkit`, and all of the action API functions defined in `ckan.logic.action`, should list exceptions raised in their docstrings.

This is because CKAN themes, extensions and API clients need to be able to call CKAN code without crashing, so they need to know what exceptions they should handle (and extension developers shouldn't have to understand the CKAN core source code).

2. On the other hand, **internal functions that are only used within CKAN shouldn't list exceptions in their docstrings.**

This is because it would be difficult to keep all the exception lists up to date with the actual code behaviour, so the docstrings would become more misleading than useful.

3. **Code should only raise exceptions from within its allowed set.**

Each module in CKAN has a set of zero or more exceptions, defined somewhere near the module, that code in that module is allowed to raise. For example `ckan/logic/__init__.py` defines a number of exception types for code in `ckan/logic/` to use. CKAN code should never raise exceptions types defined elsewhere in CKAN, in third-party code or in the Python standard library.

4. **All code should catch any exceptions raised by called functions**, and either handle the exception, re-raise the exception (if it's from the code's set of allowed exception types), or wrap the exception in an allowed exception type and re-raise it.

This is to make it easy for a CKAN core developer to look at the source code of an internal function, scan it for the keyword `raise`, and see what types of exception the function may raise, so they know what exceptions they need to catch if they're going to call the function. Developers shouldn't have to read the source of all the functions that a function calls (and the functions they call...) to find out what exceptions they need to catch to call a function without crashing.

Todo

Insert examples of how to re-raise and how to wrap-and-re-raise an exception.

Use `:raises:` to document exceptions raised by public functions. The docstring should say what type of exception is raised and under what conditions. Use `:py:class:` to reference exception types. For example:

```
def member_list(context, data_dict=None):
    '''Return the members of a group.

    ... (parameters and return values documented here) ...

    :raises: :py:class:`ckan.logic.NotFound`: if the group doesn't exist
    '''
```

Sphinx field lists

Use [Sphinx field lists](#) for documenting the parameters, exceptions and returns of functions:

- Use `:param` and `:type` to describe each parameter
- Use `:returns` and `:rtype` to describe each return
- Use `:raises` to describe each exception raised

Example of a short docstring:

```
@property
def packages(self):
    '''Return a list of all packages that have this tag, sorted by name.

    :rtype: list of ckan.model.package.Package objects
    '''
```

Example of a longer docstring:

```
@classmethod
def search_by_name(cls, search_term, vocab_id_or_name=None):
    '''Return all tags whose names contain a given string.

    By default only free tags (tags which do not belong to any vocabulary)
    are returned. If the optional argument ``vocab_id_or_name`` is given
    then only tags from that vocabulary are returned.

    :param search_term: the string to search for in the tag names
    :type search_term: string
    :param vocab_id_or_name: the id or name of the vocabulary to look in
    '''
```

```
(optional, default: None)
:type vocab_id_or_name: string

:returns: a list of tags that match the search term
:rtype: list of ckan.model.tag.Tag objects

'''
```

The phrases that follow `:param foo:`, `:type foo:`, or `:returns:` should not start with capital letters or end with full stops. These should be short phrases and not full sentences. If more detail is required put it in the function description instead.

Indicate optional arguments by ending their descriptions with `(optional)` in brackets. Where relevant also indicate the default value: `(optional, default: 5)`.

You can also use a little inline [reStructuredText markup](#) in docstrings, e.g. `*stars` for emphasis* or ```double-backticks for literal text```

Action API docstrings

Docstrings from CKAN's action API are processed with [autodoc](#) and included in the API chapter of CKAN's documentation. The intended audience of these docstrings is users of the CKAN API and not (just) CKAN core developers.

In the Python source each API function has the same two arguments (`context` and `data_dict`), but the docstrings should document the keys that the functions read from `data_dict` and not `context` and `data_dict` themselves, as this is what the user has to POST in the JSON dict when calling the API.

Where practical, it's helpful to give examples of param and return values in API docstrings.

CKAN datasets used to be called packages and the old name still appears in the source, e.g. in function names like `package_list()`. When documenting functions like this write `dataset` not `package`, but the first time you do this put `package` after it in brackets to avoid any confusion, e.g.

```
def package_show(context, data_dict):
    '''Return the metadata of a dataset (package) and its resources.
```

Example of a `ckan.logic.action` API docstring:

```
def vocabulary_create(context, data_dict):
    '''Create a new tag vocabulary.

    You must be a sysadmin to create vocabularies.

    :param name: the name of the new vocabulary, e.g. ``'Genre'``
    :type name: string
    :param tags: the new tags to add to the new vocabulary, for the format of
                  tag dictionaries see ``tag_create()``
    :type tags: list of tag dictionaries

    :returns: the newly-created vocabulary
    :rtype: dictionary

    '''
```

Some helpful tools for Python code quality

There are various tools that can help you to check your Python code for PEP8 conformance and general code quality. We recommend using them.

- `pep8` checks your Python code against some of the style conventions in PEP 8. As mentioned above, only perform style clean-ups on master to help avoid spurious merge conflicts.
- `pylint` analyzes Python source code looking for bugs and signs of poor quality.
- `pyflakes` also analyzes Python programs to detect errors.
- `flake8` combines both `pep8` and `pyflakes` into a single tool.
- `Syntastic` is a Vim plugin with support for `flake8`, `pyflakes` and `pylint`.

String internationalization

All user-facing Strings in CKAN Python, JavaScript and Jinja2 code should be internationalized, so that our translators can then localize the strings for each of the many languages that CKAN supports. This guide shows CKAN developers how to internationalize strings, and what to look for regarding string internationalization when reviewing a pull request.

Note: *Internationalization* (or *i18n*) is the process of marking strings for translation, so that the strings can be extracted from the source code and given to translators. *Localization* (*l10n*) is the process of translating the marked strings into different languages.

See also:

Translating CKAN If you want to translate CKAN, this page documents the process that translators follow to localize CKAN into different languages.

Doing a CKAN release The processes for extracting internationalized strings from CKAN and uploading them to Transifex to be translated, and for downloading the translations from Transifex and loading them into CKAN to be displayed are documented on this page.

Note: Much of the existing code in CKAN was written before we had these guidelines, so it doesn't always do things as described on this page. When writing new code you should follow the guidelines on this page, not the existing code.

Internationalizing strings in Jinja2 templates

Most user-visible strings should be in the Jinja2 templates, rather than in Python or JavaScript code. This doesn't really matter to translators, but it's good for the code to separate logic and content. Of course this isn't always possible. For example when error messages are delivered through the API, there's no Jinja2 template involved.

The preferred way to internationalize strings in Jinja2 templates is by using the `trans` tag from Jinja2's *i18n* extension, which is available to all CKAN core and extension templates and snippets.

Most of the following examples are taken from the Jinja2 docs.

To internationalize a string put it inside a `{% trans %}` tag:

```
<p>{% trans %}This paragraph is translatable.{% endtrans %}</p>
```

You can also use variables from the template's namespace inside a `{% trans %}`:

```
<p>{% trans %}Hello {{ user }}!{% endtrans %}</p>
```

(Only variable tags are allowed inside `trans` tags, not statements.)

You can pass one or more arguments to the `{% trans %}` tag to bind variable names for use within the tag:

```
<p>{% trans user=user.username %}Hello {{ user }}!{% endtrans %}</p>

{% trans book_title=book.title, author=author.name %}
This is {{ book_title }} by {{ author }}
{% endtrans %}
```

To handle different singular and plural forms of a string, use a `{% pluralize %}` tag:

```
{% trans count=list|length %}
There is {{ count }} {{ name }} object.
{% pluralize %}
There are {{ count }} {{ name }} objects.
{% endtrans %}
```

(In English the first string will be rendered if `count` is 1, the second otherwise. For other languages translators will be able to provide their own strings for different values of `count`.)

The first variable in the block (`count` in the example above) is used to determine which of the singular or plural forms to use. Alternatively you can explicitly specify which variable to use:

```
{% trans ..., user_count=users|length %}
...
{% pluralize user_count %}
...
{% endtrans %}
```

The `{% trans %}` tag is preferable, but if you need to pluralize a string within a Jinja2 expression you can use the `__()` and `ungettext()` functions:

```
{% set hello = __('Hello World!') %}
```

To use variables in strings, use Python [format string syntax](#) and then call the `.format()` method on the string that `__()` returns:

```
{% set hello = __('Hello {name}!').format(name=user.name) %}
```

Singular and plural forms are handled by `ungettext()`:

```
{% set text = ungettext(
    '{num} apple', '{num} apples', num_apples).format(num=num_apples) %}
```

Note: There are also `gettext()` and `ngettext()` functions available to templates, but we recommend using `__()` and `ungettext()` for consistency with CKAN's Python code. This deviates from the Jinja2 docs, which do use `gettext()` and `ngettext()`.

`__()` is not an alias for `gettext()` in CKAN's Jinja2 templates, `__()` is the function provided by Pylons, whereas `gettext()` is the version provided by Jinja2, their behaviors are not exactly the same.

Internationalizing strings in Python code

CKAN uses the `__()` and `ungettext()` functions from the [pylons.i18n.translation](#) module to internationalize strings in Python code.

Core CKAN modules should import `__()` and `ungettext()` from `ckan.common`, i.e. from `ckan.common` import `__, ungettext` (don't import `pylons.i18n.translation.__()` directly, for example).

CKAN plugins should import `ckan.plugins.toolkit` and use `ckan.plugins.toolkit._()` and `ckan.plugins.toolkit.ungettext()`, i.e. do import `ckan.plugins.toolkit` as `toolkit` and then use `toolkit._()` and `toolkit.ungettext()` (see [Plugins toolkit reference](#)).

To internationalize a string pass it to the `_()` function:

```
my_string = _("This paragraph is translatable.")
```

To use variables in a string, call the `.format()` method on the translated string that `_()` returns:

```
hello = _("Hello {user}!").format(user=user.name)
```

```
book_description = _("This is { book_title } by { author }").format(
    book_title=book.title, author=author.name)
```

To handle different plural and singular forms of a string, use `ungettext()`:

```
translated_string = ungettext(
    "There is {count} {name} object.",
    "There are {count} {name} objects.",
    num_objects).format(count=count, name=name)
```

Internationalizing strings in JavaScript code

Todo

General guidelines for internationalizing strings

Below are some guidelines to follow when marking your strings for translation. These apply to strings in Jinja2 templates or in Python or JavaScript code. These are mostly meant to make life easier for translators, and help to improve the quality of CKAN's translations:

- Leave as much HTML and other code out of the translation string as possible.

For example, don't include surrounding `<p>...</p>` tags in the marked string. These aren't necessary for the translator to do the translation, and if the translator accidentally changes them in the translation string the HTML will be broken.

Good:

```
<p>{% trans %}Don't put HTML tags inside translatable strings{% endtrans %}</p>
```

Bad (`<p>` tags don't need to be in the translation string):

```
mystring = _("<p>Don't put HTML tags inside translatable strings</p>")
```

- But don't split a string into separate strings.

Translators need as much context as possible to translate strings well, and if you split a string up into separate strings and mark each for translation separately, translators must translate each of these separate strings in isolation. Also, some languages may need to change the order of words in a sentence or even change the order of sentences in a paragraph, splitting into separate strings makes assumptions about word order.

It's better to leave HTML tags or other code in strings than to split a string. For example, it's often best to leave HTML `<a>` tags in rather than split a string.

Good:

```
_("Don't split a string containing some <b>markup</b> into separate strings.")
```

Bad (text will be difficult to translate or untranslatable):

```
_("Don't split a string containing some ") + "<b>" + _("markup") + </b> + _("into separate strings")
```

- You can split long strings over multiple lines using parentheses to avoid long lines, Python will concatenate them into a single string:

Good:

```
_("This is a really long string that would just make this line far too "
  "long to fit in the window")
```

- Leave unnecessary whitespace out of translatable strings, but do put punctuation into translatable strings.
- Try not to make translators translate strings that don't need to be translated.

For example, 'legacy_templates' is the name of a directory, it doesn't need to be marked for translation.

- Mark singular and plural forms of strings correctly.

In Jinja2 templates this means using `{% trans %}` and `{% pluralize %}` or `ungettext()`. In Python it means using `ungettext()`. See above for examples.

Singular and plural forms work differently in different languages. For example English has singular and plural nouns, but Slovenian has singular, dual and plural.

Good:

```
num_people = 4
translated_string = ungettext(
    'There is one person here',
    'There are {num_people} people here',
    num_people).format(num_people=num_people)
```

Bad (this assumes that all languages have the same plural forms as English):

```
if num_people == 1:
    translated_string = _('There is one person here')
else:
    translated_string = _(
        'There are {num_people} people here'.format(num_people=num_people))
```

- Don't use old-style `%s` string formatting in Python, use the new `.format()` method instead.

Strings formatted with `.format()` give translators more context. The `.format()` method is also more expressive, and is the preferred way to format strings in Python 3.

Good:

```
"Welcome to {site_title}".format(site_title=site_title)
```

Bad (not enough context for translators):

```
"Welcome to %s" % site_title
```

- Use descriptive names for replacement fields in strings.

This gives translators more context.

Good:

```
"Welcome to {site_title}".format(site_title=site_title)
```

Bad (not enough context for translators):

```
"Welcome to {0}".format(site_title)
```

Worse (doesn't work in Python 2.6):

```
"Welcome to {}".format(site_title)
```

- Use `TRANSLATORS:` comments to provide extra context for translators for difficult to find, very short, or obscure strings.

For example, in Python:

```
# TRANSLATORS: This is a helpful comment.
_("This is an ambiguous string")
```

In Jinja2:

```
{# TRANSLATORS: This heading is displayed on the user's profile page. #}
<h1>{% trans %}Heading{% endtrans %}</h1>
```

These comments end up in the `ckan.pot` file and translators will see them when they're translating the strings (Transifex shows them, for example).

Note: In both Python and Jinja2, the comment must be on the line before the line with the `_()`, `ungettext()` or `{% trans %}`, and must start with the exact string `TRANSLATORS:` (in upper-case and with the colon). This string is configured in `setup.cfg`.

Todo

Example of leaving a translator comment in JavaScript. Probably `// TRANSLATORS: This is a helpful comment` will work.

Todo

Explain how to use *message contexts*, where the same exact string may appear in two different places in the UI but have different meanings.

For example “filter” can be a noun or a verb in English, and may need two different translations in another language. Currently if the string `_("filter")` appears in different places in CKAN this will only produce one string to be translated in the `ckan.pot` file.

I think the right way to handle this with `gettext` is using `msgctxt`, but it looks like `babel` doesn't support it yet.

Todo

Explain how we internationalize dates, currencies and numbers (e.g. different positioning and separators used for decimal points in different languages).

Testing coding standards

All new code, or changes to existing code, should have new or updated tests before being merged into master. This document gives some guidelines for developers who are writing tests or reviewing code for CKAN.

Transitioning from legacy to new tests

CKAN is an old code base with a large legacy test suite in `ckan.tests.legacy`. The legacy tests are difficult to maintain and extend, but are too many to be replaced all at once in a single effort. So we're following this strategy:

1. A new test suite has been started in `ckan.tests`.
2. For now, we'll run both the legacy tests and the new tests before merging something into the master branch.
3. Whenever we add new code, or change existing code, we'll add new-style tests for it.
4. If you change the behavior of some code and break some legacy tests, consider adding new tests for that code and deleting the legacy tests, rather than updating the legacy tests.
5. Now and then, we'll write a set of new tests to cover part of the code, and delete the relevant legacy tests. For example if you want to refactor some code that doesn't have good tests, write a set of new-style tests for it first, refactor, then delete the relevant legacy tests.

In this way we can incrementally extend the new tests to cover CKAN one "island of code" at a time, and eventually we can delete the legacy `ckan.tests` directory entirely.

Guidelines for writing new-style tests

We want the tests in `ckan.tests` to be:

Fast

- Don't share setup code between tests (e.g. in test class `setup()` or `setup_class()` methods, saved against the `self` attribute of test classes, or in test helper modules).

Instead write helper functions that create test objects and return them, and have each test method call just the helpers it needs to do the setup that it needs.

- Where appropriate, use the `mock` library to avoid pulling in other parts of CKAN (especially the database), see *Mocking: the mock library*.

Independent

- Each test module, class and method should be able to be run on its own.
- Tests shouldn't be tightly coupled to each other, changing a test shouldn't affect other tests.

Clear It should be quick and easy to see what went wrong when a test fails, or to see what a test does and how it works if you have to debug or update a test. If you think the test or helper method isn't clear by itself, add docstrings.

You shouldn't have to figure out what a complex test method does, or go and look up a lot of code in other files to understand a test method.

- Tests should follow the canonical form for a unit test, see *Recipe for a test method*.
- Write lots of small, simple test methods not a few big, complex tests.
- Each test method should test just One Thing.
- The name of a test method should clearly explain the intent of the test. See *Naming test methods*.

Easy to find It should be easy to know where to add new tests for some new or changed code, or to find the existing tests for some code.

- See *How should tests be organized?*
- See *Naming test methods*.

Easy to write Writing lots of small, clear and simple tests that all follow similar recipes and organization should make tests easy to write, as well as easy to read.

The follow sections give some more specific guidelines and tips for writing CKAN tests.

How should tests be organized?

The organization of test modules in `ckan.tests` mirrors the organization of the source modules in `ckan`:

```
ckan/
  tests/
    controllers/
      test_package.py <-- Tests for ckan/controllers/package.py
      ...
    lib/
      test_helpers.py <-- Tests for ckan/lib/helpers.py
      ...
    logic/
      action/
        test_get.py
        ...
      auth/
        test_get.py
        ...
      test_converters.py
      test_validators.py
    migration/
      versions/
        test_001_add_existing_tables.py
        ...
    model/
      test_package.py
      ...
    ...
```

There are a few exceptional test modules that don't fit into this structure, for example PEP8 tests and coding standards tests. These modules can just go in the top-level `ckan/tests/` directory. There shouldn't be too many of these.

Naming test methods

The name of a test method should clearly explain the intent of the test.

Test method names are printed out when tests fail, so the user can often see what went wrong without having to look into the test file. When they do need to look into the file to debug or update a test, the test name helps to clarify the test.

Do this even if it means your method name gets really long, since we don't write code that calls our test methods there's no advantage to having short test method names.

Some modules in CKAN contain large numbers of loosely related functions. For example, `ckan.logic.action.update` contains all functions for updating things in CKAN. This means that `ckan.tests.logic.action.test_update` is going to contain an even larger number of test functions.

So as well as the name of each test method explaining the intent of the test, tests should be grouped by a test class that aggregates tests against a model entity or action type, for instance:

```
class TestPackageCreate(object):
    # ...
    def test_it_validates_name(self):
        # ...

    def test_it_validates_url(self):
        # ...

class TestResourceCreate(object):
    # ...
    def test_it_validates_package_id(self):
        # ...

# ...
```

Good test names:

- `TestUserUpdate.test_update_with_id_that_does_not_exist`
- `TestUserUpdate.test_update_with_no_id`
- `TestUserUpdate.test_update_with_invalid_name`

Bad test names:

- `test_user_update`
- `test_update_pkg_1`
- `test_package`

Recipe for a test method

The [Pylons Unit Testing Guidelines](#) give the following recipe for all unit test methods to follow:

1. Set up the preconditions for the method / function being tested.
2. Call the method / function exactly one time, passing in the values established in the first step.
3. Make assertions about the return value, and / or any side effects.
4. Do absolutely nothing else.

Most CKAN tests should follow this form. Here's an example of a simple action function test demonstrating the recipe:

```
def test_user_update_name(self):
    '''Test that updating a user's name works successfully.'''

    # The canonical form of a test has four steps:
    # 1. Setup any preconditions needed for the test.
    # 2. Call the function that's being tested, once only.
    # 3. Make assertions about the return value and/or side-effects of
    #    of the function that's being tested.
    # 4. Do nothing else!

    # 1. Setup.
    user = factories.User()

    # 2. Call the function that's being tested, once only.
    # FIXME we have to pass the email address and password to user_update
```



```

# even though we're not updating those fields, otherwise validation
# fails.
helpers.call_action('user_update', id=user['name'],
                    email=user['email'],
                    password=factories.User.attributes()['password'],
                    name='updated',
                    )

# 3. Make assertions about the return value and/or side-effects.
updated_user = helpers.call_action('user_show', id=user['id'])
# Note that we check just the field we were trying to update, not the
# entire dict, only assert what we're actually testing.
assert updated_user['name'] == 'updated'

# 4. Do nothing else!

```

One common exception is when you want to use a `for` loop to call the function being tested multiple times, passing it lots of different arguments that should all produce the same return value and/or side effects. For example, this test from `ckan.tests.logic.action.test_update`:

```

def test_user_update_with_invalid_name(self):
    user = factories.User()

    invalid_names = ('', 'a', False, 0, -1, 23, 'new', 'edit', 'search',
                    'a' * 200, 'Hi!', 'i++%')
    for name in invalid_names:
        user['name'] = name
        assert_raises(logic.ValidationError,
                      helpers.call_action, 'user_update',
                      **user)

```

The behavior of `user_update()` is the same for every invalid value. We do want to test `user_update()` with lots of different invalid names, but we obviously don't want to write a dozen separate test methods that are all the same apart from the value used for the invalid user name. We don't really want to define a helper method and a dozen test methods that call it either. So we use a simple loop. Technically this test calls the function being tested more than once, but there's only one line of code that calls it.

How detailed should tests be?

Generally, what we're trying to do is test the *interfaces* between modules in a way that supports modularization: if you change the code within a function, method, class or module, if you don't break any of that code's tests you should be able to expect that CKAN as a whole will not be broken.

As a general guideline, the tests for a function or method should:

- Test for success:
 - Test the function with typical, valid input values
 - Test with valid, edge-case inputs
 - If the function has multiple parameters, test them in different combinations
- Test for failure:
 - Test that the function fails correctly (e.g. raises the expected type of exception) when given likely invalid inputs (for example, if the user passes an invalid `user_id` as a parameter)
 - Test that the function fails correctly when given bizarre input

- Test that the function behaves correctly when given unicode characters as input
- Cover the interface of the function: test all the parameters and features of the function

Creating test objects: `ckan.tests.factories`

This is a collection of factory classes for building CKAN users, datasets, etc.

These are meant to be used by tests to create any objects that are needed for the tests. They're written using `factory_boy`:

<http://factoryboy.readthedocs.org/en/latest/>

These are not meant to be used for the actual testing, e.g. if you're writing a test for the `user_create()` function then call `call_action()`, don't test it via the `User` factory below.

Usage:

```
# Create a user with the factory's default attributes, and get back a
# user dict:
user_dict = factories.User()

# You can create a second user the same way. For attributes that can't be
# the same (e.g. you can't have two users with the same name) a new value
# will be generated each time you use the factory:
another_user_dict = factories.User()

# Create a user and specify your own user name and email (this works
# with any params that CKAN's user_create() accepts):
custom_user_dict = factories.User(name='bob', email='bob@bob.com')

# Get a user dict containing the attributes (name, email, password, etc.)
# that the factory would use to create a user, but without actually
# creating the user in CKAN:
user_attributes_dict = factories.User.attributes()

# If you later want to create a user using these attributes, just pass them
# to the factory:
user = factories.User(**user_attributes_dict)
```

class `ckan.tests.factories.User`
A factory class for creating CKAN users.

class `ckan.tests.factories.Resource`
A factory class for creating CKAN resources.

class `ckan.tests.factories.ResourceView`
A factory class for creating CKAN resource views.

Note: if you use this factory, you need to load the `image_view` plugin on your test class (and unload it later), otherwise you will get an error.

Example:

```
class TestSomethingWithResourceViews(object):
    @classmethod
    def setup_class(cls):
        if not p.plugin_loaded('image_view'):
            p.load('image_view')
```

```
@classmethod
def teardown_class(cls):
    p.unload('image_view')
```

class `ckan.tests.factories.Sysadmin`
A factory class for creating sysadmin users.

class `ckan.tests.factories.Group`
A factory class for creating CKAN groups.

class `ckan.tests.factories.Organization`
A factory class for creating CKAN organizations.

class `ckan.tests.factories.Related`
A factory class for creating related items.

class `ckan.tests.factories.Dataset`
A factory class for creating CKAN datasets.

class `ckan.tests.factories.MockUser`
A factory class for creating mock CKAN users using the mock library.

FACTORY_FOR
alias of `MagicMock`

class `ckan.tests.factories.SystemInfo`
A factory class for creating `SystemInfo` objects (config objects stored in the DB).

`ckan.tests.factories.validator_data_dict()`
Return a data dict with some arbitrary data in it, suitable to be passed to validator functions for testing.

`ckan.tests.factories.validator_errors_dict()`
Return an errors dict with some arbitrary errors in it, suitable to be passed to validator functions for testing.

class `ckan.tests.factories.Vocabulary`
A factory class for creating tag vocabularies.

Test helper functions: `ckan.tests.helpers`

This is a collection of helper functions for use in tests.

We want to avoid sharing test helper functions between test modules as much as possible, and we definitely don't want to share test fixtures between test modules, or to introduce a complex hierarchy of test class subclasses, etc.

We want to reduce the amount of “travel” that a reader needs to undertake to understand a test method – reducing the number of other files they need to go and read to understand what the test code does. And we want to avoid tightly coupling test modules to each other by having them share code.

But some test helper functions just increase the readability of tests so much and make writing tests so much easier, that it's worth having them despite the potential drawbacks.

This module is reserved for these very useful functions.

`ckan.tests.helpers.reset_db()`
Reset CKAN's database.

If a test class uses the database, then it should call this function in its `setup()` method to make sure that it has a clean database to start with (nothing left over from other test classes or from previous test runs).

If a test class doesn't use the database (and most test classes shouldn't need to) then it doesn't need to call this function.

Returns None

`ckan.tests.helpers.call_action(action_name, context=None, **kwargs)`

Call the named `ckan.logic.action` function and return the result.

This is just a nicer way for user code to call action functions, nicer than either calling the action function directly or via `ckan.logic.get_action()`.

For example:

```
user_dict = call_action('user_create', name='seanh',
                        email='seanh@seanh.com', password='pass')
```

Any keyword arguments given will be wrapped in a dict and passed to the action function as its `data_dict` argument.

Note: this skips authorization! It passes `'ignore_auth': True` to action functions in their context dicts, so the corresponding authorization functions will not be run. This is because `ckan.tests.logic.action` tests only the actions, the authorization functions are tested separately in `ckan.tests.logic.auth`. See the [testing guidelines](#) for more info.

This function should eventually be moved to `ckan.logic.call_action()` and the current `ckan.logic.get_action()` function should be deprecated. The tests may still need their own wrapper function for `ckan.logic.call_action()`, e.g. to insert `'ignore_auth': True` into the context dict.

Parameters

- **action_name** (*string*) – the name of the action function to call, e.g. `'user_update'`
- **context** (*dict*) – the context dict to pass to the action function (optional, if no context is given a default one will be supplied)

Returns the dict or other value that the action function returns

`ckan.tests.helpers.call_auth(auth_name, context, **kwargs)`

Call the named `ckan.logic.auth` function and return the result.

This is just a convenience function for tests in `ckan.tests.logic.auth` to use.

Usage:

```
result = helpers.call_auth('user_update', context=context,
                           id='some_user_id',
                           name='updated_user_name')
```

Parameters

- **auth_name** (*string*) – the name of the auth function to call, e.g. `'user_update'`
- **context** (*dict*) – the context dict to pass to the auth function, must contain `'user'` and `'model'` keys, e.g. `{ 'user': 'fred', 'model': my_mock_model_object }`

Returns the dict that the auth function returns, e.g. `{ 'success': True }` or `{ 'success': False, msg: '...' }` or just `{ 'success': False }`

Return type dict

class `ckan.tests.helpers.FunctionalTestBase`

A base class for functional test classes to inherit from.

Allows configuration changes by overriding `_apply_config_changes` and resetting the CKAN config after your test class has run. It creates a `webtest.TestApp` at `self.app` for your class to use to make HTTP requests to the CKAN web UI or API.

If you're overriding methods that this class provides, like `setup_class()` and `teardown_class()`, make sure to use `super()` to call this class's methods at the top of yours!

setup()

Reset the database and clear the search indexes.

```
ckan.tests.helpers.submit_and_follow(app, form, extra_environ, name=None, value=None,
                                     **args)
```

Call `webtest_submit` with `name/value` passed expecting a redirect and return the response from following that redirect.

```
ckan.tests.helpers.webtest_submit(form, name=None, index=None, value=None, **args)
```

backported version of `webtest.Form.submit` that actually works for submitting with different submit buttons.

We're stuck on an old version of `webtest` because we're stuck on an old version of `webob` because we're stuck on an old version of `Pylons`. This prolongs our suffering, but on the bright side it lets us have functional tests that work.

```
ckan.tests.helpers.webtest_submit_fields(form, name=None, index=None, submit_value=None)
```

backported version of `webtest.Form.submit_fields` that actually works for submitting with different submit buttons.

```
ckan.tests.helpers.change_config(key, value)
```

Decorator to temporarily changes `Pylons`' config to a new value

This allows you to easily create tests that need specific config values to be set, making sure it'll be reverted to what it was originally, after your test is run.

Usage:

```
@helpers.change_config('ckan.site_title', 'My Test CKAN')
def test_ckan_site_title(self):
    assert pylons.config['ckan.site_title'] == 'My Test CKAN'
```

Parameters

- **key** (*string*) – the config key to be changed, e.g. `'ckan.site_title'`
- **value** (*string*) – the new config key's value, e.g. `'My Test CKAN'`

Mocking: the `mock` library

We use the [mock library](#) to replace parts of CKAN with mock objects. This allows a CKAN function to be tested independently of other parts of CKAN or third-party libraries that the function uses. This generally makes the test simpler and faster (especially when `ckan.model` is mocked out so that the tests don't touch the database). With mock objects we can also make assertions about what methods the function called on the mock object and with which arguments.

Note: Overuse of mocking is discouraged as it can make tests difficult to understand and maintain. Mocking can be useful and make tests both faster and simpler when used appropriately. Some rules of thumb:

- Don't mock out more than one or two objects in a single test method.

- Don't use mocking in more functional-style tests. For example the action function tests in `ckan.tests.logic.action` and the frontend tests in `ckan.tests.controllers` are functional tests, and probably shouldn't do any mocking.
- Do use mocking in more unit-style tests. For example the authorization function tests in `ckan.tests.logic.auth`, the converter and validator tests in `ckan.tests.logic.auth`, and most (all?) lib tests in `ckan.tests.lib` are unit tests and should use mocking when necessary (often it's possible to unit test a method in isolation from other CKAN code without doing any mocking, which is ideal).

In these kind of tests we can often mock one or two objects in a simple and easy to understand way, and make the test both simpler and faster.

A mock object is a special object that allows user code to access any attribute name or call any method name (and pass any parameters) on the object, and the code will always get another mock object back:

```
>>> import mock
>>> my_mock = mock.MagicMock()
>>> my_mock.foo
<MagicMock name='mock.foo' id='56032400'>
>>> my_mock.bar
<MagicMock name='mock.bar' id='54093968'>
>>> my_mock.foobar()
<MagicMock name='mock.foobar()' id='54115664'>
>>> my_mock.foobar(1, 2, 'barfoo')
<MagicMock name='mock.foobar()' id='54115664'>
```

When a test needs a mock object to actually have some behavior besides always returning other mock objects, it can set the value of a certain attribute on the mock object, set the return value of a certain method, specify that a certain method should raise a certain exception, etc.

You should read the mock library's documentation to really understand what's going on, but here's an example of a test from `ckan.tests.logic.auth.test_update` that tests the `user_update()` authorization function and mocks out `ckan.model`:

```
def test_user_update_user_cannot_update_another_user(self):
    '''Users should not be able to update other users' accounts.'''

    # 1. Setup.

    # Make a mock ckan.model.User object, Fred.
    fred = factories.MockUser(name='fred')

    # Make a mock ckan.model object.
    mock_model = mock.MagicMock()
    # model.User.get(user_id) should return Fred.
    mock_model.User.get.return_value = fred

    # Put the mock model in the context.
    # This is easier than patching import ckan.model.
    context = {'model': mock_model}

    # The logged-in user is going to be Bob, not Fred.
    context['user'] = 'bob'

    # 2. Call the function that's being tested, once only.

    # Make Bob try to update Fred's user account.
    params = {
        'id': fred.id,
```

```

        'name': 'updated_user_name',
    }

    # 3. Make assertions about the return value and/or side-effects.

    nose.tools.assert_raises(logic.NotAuthorized, helpers.call_auth,
                             'user_update', context=context, **params)

    # 4. Do nothing else!

```

The following sections will give specific guidelines and examples for writing tests for each module in CKAN.

Note: When we say that *all* functions should have tests in the sections below, we mean all *public* functions that the module or class exports for use by other modules or classes in CKAN or by extensions or templates.

Private helper methods (with names beginning with `_`) never have to have their own tests, although they can have tests if helpful.

Writing `ckan.logic.action` tests

All action functions should have tests.

Most action function tests will be high-level tests that both test the code in the action function itself, and also indirectly test the code in `ckan.lib`, `ckan.model`, `ckan.logic.schema` etc. that the action function calls. This means that most action function tests should *not* use mocking.

Tests for action functions should use the `ckan.tests.helpers.call_action()` function to call the action functions.

One thing `call_action()` does is to add `ignore_auth: True` into the `context` dict that's passed to the action function, so that CKAN will not call the action function's authorization function. The tests for an action function *don't* need to cover authorization, because the authorization functions have their own tests in `ckan.tests.logic.auth`. But action function tests *do* need to cover validation, more on that later.

Action function tests *should* test the logic of the actions themselves, and *should* test validation (e.g. that various kinds of valid input work as expected, and invalid inputs raise the expected exceptions).

Here's an example of a simple `ckan.logic.action` test:

```

def test_user_update_name(self):
    '''Test that updating a user's name works successfully.'''

    # The canonical form of a test has four steps:
    # 1. Setup any preconditions needed for the test.
    # 2. Call the function that's being tested, once only.
    # 3. Make assertions about the return value and/or side-effects of
    #    of the function that's being tested.
    # 4. Do nothing else!

    # 1. Setup.
    user = factories.User()

    # 2. Call the function that's being tested, once only.
    # FIXME we have to pass the email address and password to user_update
    # even though we're not updating those fields, otherwise validation
    # fails.

```

```
helpers.call_action('user_update', id=user['name'],
                    email=user['email'],
                    password=factories.User.attributes()['password'],
                    name='updated',
                    )

# 3. Make assertions about the return value and/or side-effects.
updated_user = helpers.call_action('user_show', id=user['id'])
# Note that we check just the field we were trying to update, not the
# entire dict, only assert what we're actually testing.
assert updated_user['name'] == 'updated'

# 4. Do nothing else!
```

Todo

Insert the names of all tests for `ckan.logic.action.update.user_update`, for example, to show what level of detail things should be tested in.

Writing `ckan.logic.auth` tests

All auth functions should have tests.

Most auth function tests should be unit tests that test the auth function in isolation, without bringing in other parts of CKAN or touching the database. This requires using the mock library to mock `ckan.model`, see [Mocking: the mock library](#).

Tests for auth functions should use the `ckan.tests.helpers.call_auth()` function to call auth functions.

Here's an example of a simple `ckan.logic.auth` test:

```
def test_user_update_user_cannot_update_another_user(self):
    '''Users should not be able to update other users' accounts.'''

    # 1. Setup.

    # Make a mock ckan.model.User object, Fred.
    fred = factories.MockUser(name='fred')

    # Make a mock ckan.model object.
    mock_model = mock.MagicMock()
    # model.User.get(user_id) should return Fred.
    mock_model.User.get.return_value = fred

    # Put the mock model in the context.
    # This is easier than patching import ckan.model.
    context = {'model': mock_model}

    # The logged-in user is going to be Bob, not Fred.
    context['user'] = 'bob'

    # 2. Call the function that's being tested, once only.

    # Make Bob try to update Fred's user account.
    params = {
        'id': fred.id,
        'name': 'updated_user_name',
```



```

}

# 3. Make assertions about the return value and/or side-effects.

nose.tools.assert_raises(logic.NotAuthorized, helpers.call_auth,
                          'user_update', context=context, **params)

# 4. Do nothing else!

```

Writing converter and validator tests

All converter and validator functions should have unit tests.

Although these converter and validator functions are tested indirectly by the action function tests, this may not catch all the converters and validators and all their options, and converters and validators are not only used by the action functions but are also available to plugins. Having unit tests will also help to clarify the intended behavior of each converter and validator.

CKAN's action functions call `ckan.lib.navl.dictization_functions.validate()` to validate data posted by the user. Each action function passes a schema from `ckan.logic.schema` to `validate()`. The schema gives `validate()` lists of validation and conversion functions to apply to the user data. These validation and conversion functions are defined in `ckan.logic.validators`, `ckan.logic.converters` and `ckan.lib.navl.validators`.

Most validator and converter tests should be unit tests that test the validator or converter function in isolation, without bringing in other parts of CKAN or touching the database. This requires using the `mock` library to mock `ckan.model`, see [Mocking: the mock library](#).

When testing validators, we often want to make the same assertions in many tests: assert that the validator didn't modify the data dict, assert that the validator didn't modify the `errors` dict, assert that the validator raised `Invalid`, etc. Decorator functions are defined at the top of validator test modules like `ckan.tests.logic.test_validators` to make these common asserts easy. To use one of these decorators you have to:

1. Define a nested function inside your test method, that simply calls the validator function that you're trying to test.
2. Apply the decorators that you want to this nested function.
3. Call the nested function.

Here's an example of a simple validator test that uses this technique:

```

def test_user_name_validator_with_non_string_value(self):
    '''user_name_validator() should raise Invalid if given a non-string
    value.

    '''
    non_string_values = [
        13,
        23.7,
        100L,
        1.0j,
        None,
        True,
        False,
        ('a', 2, False),
        [13, None, True],
    ]

```

```
{'foo': 'bar'},
    lambda x: x ** 2,
]

# Mock ckan.model.
mock_model = mock.MagicMock()
# model.User.get(some_user_id) needs to return None for this test.
mock_model.User.get.return_value = None

key = ('name',)
for non_string_value in non_string_values:
    data = factories.validator_data_dict()
    data[key] = non_string_value
    errors = factories.validator_errors_dict()
    errors[key] = []

    @t.does_not_modify_data_dict
    @raises_Invalid
    def call_validator(*args, **kwargs):
        return validators.user_name_validator(*args, **kwargs)
    call_validator(key, data, errors, context={'model': mock_model})
```

No tests for `ckan.logic.schema.py`

We *don't* write tests for the schemas defined in `ckan.logic.schema`. The validation done by the schemas is instead tested indirectly by the action function tests. The reason for this is that CKAN actually does validation in multiple places: some validation is done using schemas, some validation is done in the action functions themselves, some is done in dictization, and some in the model. By testing all the different valid and invalid inputs at the action function level, we catch it all in one place.

Writing `ckan.controllers` tests

Controller tests probably shouldn't use mocking.

Todo

Write the tests for one controller, figuring out the best way to write controller tests. Then fill in this guidelines section, using the first set of controller tests as an example.

Some things have been decided already:

- All controller methods should have tests
- Controller tests should be high-level tests that work by posting simulated HTTP requests to CKAN URLs and testing the response. So the controller tests are also testing CKAN's templates and rendering - these are CKAN's front-end tests.

For example, maybe we use a webtests testapp and then use beautiful soup to parse the HTML?

- In general the tests for a controller shouldn't need to be too detailed, because there shouldn't be a lot of complicated logic and code in controller classes. The logic should be handled in other places such as `ckan.logic` and `ckan.lib`, where it can be tested easily and also shared with other code.
- The tests for a controller should:
 - Make sure that the template renders without crashing.

- Test that the page contents seem basically correct, or test certain important elements in the page contents (but don't do too much HTML parsing).
- Test that submitting any forms on the page works without crashing and has the expected side-effects.
- When asserting side-effects after submitting a form, controller tests should use the `ckan.tests.helpers.call_action()` function. For example after creating a new user by submitting the new user form, a test could call the `user_show()` action function to verify that the user was created with the correct values.

Warning: Some CKAN controllers *do* contain a lot of complicated logic code. These controllers should be refactored to move the logic into `ckan.logic` or `ckan.lib` where it can be tested easily. Unfortunately in cases like this it may be necessary to write a lot of controller tests to get this code's behavior into a test harness before it can be safely refactored.

Writing `ckan.model` tests

All model methods should have tests.

Todo

Write the tests for one `ckan.model` module, figuring out the best way to write model tests. Then fill in this guidelines section, using the first set of model tests as an example.

Writing `ckan.lib` tests

All lib functions should have tests.

Todo

Write the tests for one `ckan.lib` module, figuring out the best way to write lib tests. Then fill in this guidelines section, using the first

We probably want to make these unit tests rather than high-level tests and mock out `ckan.model`, so the tests are really fast and simple.

Note that some things in lib are particularly important, e.g. the functions in `ckan.lib.helpers` are exported for templates (including extensions) to use, so all of these functions should really have tests and docstrings. It's probably worth focusing on these modules first.

Writing `ckan.plugins` tests

The plugin interfaces in `ckan.plugins.interfaces` are not directly testable because they don't contain any code, *but*:

- Each plugin interface should have an example plugin in `ckan.ckanext` and the example plugin should have its own functional tests.
- The tests for the code that calls the plugin interface methods should test that the methods are called correctly.

For example `ckan.logic.action.get.package_show()` calls `ckan.plugins.interfaces.IDatasetForm.read()` so the `package_show()` tests should include tests that `read()` is called at the right times and with the right parameters.

Everything in `ckan.plugins.toolkit` should have tests, because these functions are part of the API for extensions to use. But `toolkit` imports most of these functions from elsewhere in CKAN, so the tests should be elsewhere also, in the test modules for the modules where the functions are defined.

Other than the plugin interfaces and plugins toolkit, any other code in `ckan.plugins` should have tests.

Writing `ckan.migration` tests

All migration scripts should have tests.

Todo

Write some tests for a migration script, and then use them as an example to fill out this guidelines section.

Writing `ckan.ckanext` tests

Within extensions, follow the same guidelines as for CKAN core. For example if an extension adds an action function then the action function should have tests, etc.

Frontend development guidelines

Templating

Within CKAN 2.0 we moved out templating to use Jinja2 from Genshi. This was done to provide a more flexible, extensible and most importantly easy to understand templating language.

Some useful links to get you started.

- [Jinja2 Homepage](#)
- [Jinja2 Developer Documentation](#)
- [Jinja2 Template Documentation](#)

Legacy Templates

Existing Genshi templates have been moved to the `templates_legacy` directory and will continue to be served if no file with the same name is located in `templates`. This should ensure backward compatibility until instances are able to upgrade to the new system.

The lookup path for templates is as follows. Give the template path “user/index.html”:

1. Look in the template directory of each loaded extension.
2. Look in the `template_legacy` directory for each extension.
3. Look in the main `ckan` template directory.
4. Look in the `template_legacy` directory.

CKAN will automatically determine the template engine to use.

File Structure

The file structure for the CKAN templates is pretty much the same as before with a directory per controller and individual files per action.

With Jinja2 we also have the ability to use snippets which are small fragments of HTML code that can be pulled in to any template. These are kept in a snippets directory within the same folder as the actions that are using them. More generic snippets are added to templates/snippets.

```
templates/
  base.html          # A base template with just core HTML structure
  page.html          # A base template with default page layout
  header.html        # The site header.
  footer.html        # The site footer.
  snippets/          # A folder of generic sitewide snippets
  home/
    index.html       # Template for the index action of the home controller
    snippets/        # Snippets for the home controller
  user/
    ...
templates_legacy/
  # All ckan templates
```

Using the templating system

Jinja2 makes heavy use of template inheritance to build pages. A template for an action will tend to inherit from *page.html*:

```
{% extends "page.html" %}
```

Each parent defines a number of blocks that can be overridden to add content to the page. *page.html* defines majority of the markup for a standard page. Generally only `{% block primary_content %}` needs to be extended:

```
{% extends "page.html" %}

{% block page_content.html %}
  <h1>My page title</h1>
  <p>This content will be added to the page</p>
{% endblock %}
```

Most template pages will define enough blocks so that the extending page can customise as little or as much as required.

Internationalisation

Jinja2 provides a couple of helpers for [internationalisation](#). The most common is to use the `__()` function:

```
{% block page_content.html %}
  <h1>{{ __('My page title') }}</h1>
  <p>{{ __('This content will be added to the page') }}</p>
{% endblock %}
```

Variables can be provided using the “format” function:

```
{% block page_content.html %}
  <p>{{ __('Welcome to CKAN {name}').format(name=username) }}</p>
{% endblock %}
```

For longer multiline blocks the `{% trans %}` block can be used.

```
{% block page_content.html %}
  <p>
    {% trans name=username %}
      Welcome to CKAN {{ name }}
    {% endtrans %}
  </p>
{% endblock %}
```

Conventions

There are a few common conventions that have evolved from using the language.

Includes

Note: Includes should be avoided as they are not portable use `{% snippet %}` tags whenever possible.

Snippets of text that are included using `{% include %}` should be kept in a directory called `_snippets_`. This should be kept in the same directory as the code that uses it.

Generally we use the `{% snippet %}` helper in all theme files unless the parents context must absolutely be available to the snippet. In which case the usage should be clearly documented.

Snippets

Note: `{% snippet %}` tags should be used in favour of `h.snippet()`

Snippets are essentially middle ground between includes and macros in that they are includes that allow a specific context to be provided (includes just receive the parent context).

These should be preferred to includes at all times as they make debugging much easier.

Macros

Macros should be used very sparingly to create custom generators for very generic snippets of code. For example `macros/form.html` has macros for creating common form fields.

They should generally be avoided as they are hard to extend and customise.

Templating within extensions

When you need to add or customize a template from within an extension you need to tell CKAN that there is a template directory that it can call from. Within your `update_config` method for the extension you'll need to add a `extra_template_paths` to the config.

Custom Control Structures

We've provided a few additional control structures to make working with the templates easier. Other helpers can still be used using the `h` object as before.

ckan_extends

```
{% ckan_extends %}
```

This works in a very similar way to `{% extend %}` however it will load the next template up in the load path with the same name.

For example if you wish to remove the breadcrumb from the user profile page in your own site. You would locate the template you wish to override.

```
ckan/templates/user/read.html
```

And create a new one in your theme extension.

```
ckanext-mytheme/ckanext/mytheme/templates/user/read.html
```

In this new file you would pull in the core template using `{% ckan_extends %}`:

```
{% ckan_extends %}
```

This will now render the current user/read page but we can override any portion that we wish to change. In this case the `breadcrumb` block.

```
{% ckan_extends %}
```

```
{# Remove the breadcrumb #}
{% block breadcrumb %}{% endblock %}
```

This function works recursively and so is ideal for extensions that wish to add a small snippet of functionality to the page.

Note: `{% ckan_extend %}` only extends templates of the same name.

snippet

```
{% snippet [filepath], [arg1=arg1], [arg2=arg2]... %}
```

Snippets work very much like Jinja2's `{% include %}` except that that do not inherit the parent templates context. This means that all variables must be explicitly passed in to the snippet. This makes debugging much easier.

```
{% snippet "package/snippets/package_form.html", data=data, errors=errors %}
```

url_for

```
{% url_for [arg1=arg1], [arg2=arg2]... %}
```

Works exactly the same as `h.url_for()`:

```
<a href="{% url_for controller="home", action="index" %}">Home</a>
```

link_for

```
{% link_for text, [arg1=arg1], [arg2=arg2]... %}
```

Works exactly the same as `h.link_for()`:

```
<li>{% link_for _("Home"), controller="home", action="index" %}</li>
```

url_for_static

```
{% url_for_static path %}
```

Works exactly the same as `h.url_for_static()`:

```
<script src="{% url_for_static '/javascript/home.js' %}"></script>
```

Form Macros

For working with forms we have provided some simple macros for generating common fields. These will be suitable for most forms but anything more complicated will require the markup to be written by hand.

The macros can be imported into the page using the `{% import %}` command.

```
{% import 'macros/form.html' as form %}
```

The following fields are provided:

form.input()

Creates all the markup required for an input element. Handles matching labels to inputs, error messages and other useful elements.

<code>name</code>	- The name of the form parameter.
<code>id</code>	- The id to use on the input and label. Convention is to prefix with 'field-'.
<code>label</code>	- The human readable label.
<code>value</code>	- The value of the input.
<code>placeholder</code>	- Some placeholder text.
<code>type</code>	- The type of input eg. email, url, date (default: text).
<code>error</code>	- A list of error strings for the field or just true to highlight the field.
<code>classes</code>	- An array of classes to apply to the control-group.
<code>attrs</code>	- Dictionary of extra tag attributes
<code>is_required</code>	- Boolean of whether this input is required for the form to validate

Examples:

```
{% import 'macros/form.html' as form %}
{{ form.input('title', label=_('Title'), value=data.title, error=errors.title) }}
```

form.checkbox()

Builds a single checkbox input.

<code>name</code>	- The name of the form parameter.
<code>id</code>	- The id to use on the input and label. Convention is to prefix with 'field-'.
<code>label</code>	- The human readable label.
<code>value</code>	- The value of the input.
<code>checked</code>	- If true the checkbox will be checked
<code>error</code>	- An error string for the field or just true to highlight the field.

classes - An array of classes to apply to the control-group.
 attrs - Dictionary of extra tag attributes
 is_required - Boolean of whether this input is required for the form to validate

Example:

```
{% import 'macros/form.html' as form %}
{{ form.checkbox('remember', checked=true) }}
```

form.select()

Creates all the markup required for an select element. Handles matching labels to inputs and error messages.

A field should be a dict with a “value” key and an optional “text” key which will be displayed to the user. {"value": "my-option", "text": "My Option"}. We use a dict to easily allow extension in future should extra options be required.

name - The name of the form parameter.
 id - The id to use on the input and label. Convention is to prefix with 'field-'.
 label - The human readable label.
 options - A list/tuple of fields to be used as <options>.
 selected - The value of the selected <option>.
 error - A list of error strings for the field or just true to highlight the field.
 classes - An array of classes to apply to the control-group.
 attrs - Dictionary of extra tag attributes
 is_required - Boolean of whether this input is required for the form to validate

Examples:

```
{% import 'macros/form.html' as form %}
{{ form.select('year', label=_('Year'), options={'value': 2010, 'value': 2011}, selected=2011, error=error) }}
```

form.textarea()

Creates all the markup required for a plain textarea element. Handles matching labels to inputs, selected item and error messages.

name - The name of the form parameter.
 id - The id to use on the input and label. Convention is to prefix with 'field-'.
 label - The human readable label.
 value - The value of the input.
 placeholder - Some placeholder text.
 error - A list of error strings for the field or just true to highlight the field.
 classes - An array of classes to apply to the control-group.
 attrs - Dictionary of extra tag attributes
 is_required - Boolean of whether this input is required for the form to validate

Examples:

```
{% import 'macros/form.html' as form %}
{{ form.textarea('desc', id='field-description', label=_('Description'), value=data.desc, error=error) }}
```

form.markdown()

Creates all the markup required for a Markdown textarea element. Handles matching labels to inputs, selected item and error messages.

name	- The name of the form parameter.
id	- The id to use on the input and label. Convention is to prefix with 'field-'.
label	- The human readable label.
value	- The value of the input.
placeholder	- Some placeholder text.
error	- A list of error strings for the field or just true to highlight the field.
classes	- An array of classes to apply to the control-group.
attrs	- Dictionary of extra tag attributes
is_required	- Boolean of whether this input is required for the form to validate

Examples:

```
{% import 'macros/form.html' as form %}
{{ form.markdown('desc', id='field-description', label=_('Description'), value=data.desc, error=error) }}
```

form.prepend()

Creates all the markup required for an input element with a prefixed segment. These are useful for showing url slugs and other fields where the input information forms only part of the saved data.

name	- The name of the form parameter.
id	- The id to use on the input and label. Convention is to prefix with 'field-'.
label	- The human readable label.
prepend	- The text that will be prepended before the input.
value	- The value of the input. which will use the name key as the value.
placeholder	- Some placeholder text.
error	- A list of error strings for the field or just true to highlight the field.
classes	- An array of classes to apply to the control-group.
attrs	- Dictionary of extra tag attributes
is_required	- Boolean of whether this input is required for the form to validate

Examples:

```
{% import 'macros/form.html' as form %}
{{ form.prepend('slug', id='field-slug', prepend='/dataset/', label=_('Slug'), value=data.slug, error=error) }}
```

form.custom()

Creates all the markup required for an custom key/value input. These are usually used to let the user provide custom meta data. Each “field” has three inputs one for the key, one for the value and a checkbox to remove it. So the arguments for this macro are nearly all tuples containing values for the (key, value, delete) fields respectively.

name	- A tuple of names for the three fields.
id	- An id string to be used for each input.
label	- The human readable label for the main label.
values	- A tuple of values for the (key, value, delete) fields. If delete is truthy the checkbox will be checked.
placeholder	- A tuple of placeholder text for the (key, value) fields.
error	- A list of error strings for the field or just true to highlight the field.
classes	- An array of classes to apply to the control-group.

`attrs` - Dictionary of extra tag attributes
`is_required` - Boolean of whether this input is required for the form to validate

Examples:

```
{% import 'macros/form.html' as form %}
{{ form.custom(
    names=('custom_key', 'custom_value', 'custom_deleted'),
    id='field-custom',
    label=_('Custom Field'),
    values=(extra.key, extra.value, extra.deleted),
    error='')
}}
```

`form.autoform()`

Builds a form from the supplied `form_info` list/tuple.

`form_info` - A list of dicts describing the form field to build.
`data` - The form data object.
`errors` - The form errors object.
`error_summary` - The form errors object.

Example

```
{% set form_info = [
    {'name': 'ckan.site_title', 'control': 'input', 'label': _('Site Title'), 'placeholder': ''},
    {'name': 'ckan.main_css', 'control': 'select', 'options': styles, 'label': _('Style'), 'placeholder': ''},
    {'name': 'ckan.site_description', 'control': 'input', 'label': _('Site Tag Line'), 'placeholder': ''},
    {'name': 'ckan.site_logo', 'control': 'input', 'label': _('Site Tag Logo'), 'placeholder': ''},
    {'name': 'ckan.site_about', 'control': 'markdown', 'label': _('About'), 'placeholder': _('About p...')},
    {'name': 'ckan.site_intro_text', 'control': 'markdown', 'label': _('Intro Text'), 'placeholder': ''},
    {'name': 'ckan.site_custom_css', 'control': 'textarea', 'label': _('Custom CSS'), 'placeholder': ''}
] %}

{% import 'macros/form.html' as form %}
{{ form.autoform(form_info, data, errors) }}
```

Resources

Note: Resources are only supported in the new Jinja2 style templates in CKAN 2.0 and above.

Resources are .css and .js files that may be included in an html page. Resources are included in the page by using the `{% resource %}` tag and CKAN uses [Fanstatic](#) to serve these resources.

```
{% resource 'library_name/resource_name' %}
```

Resources are grouped into libraries and the full resource name consists of `<library name>/<resource name>`. For example:

```
{% resource 'my_fanstatic_library/my_javascript_file.js' %}
```

It is important to note that these resources will be added to the page as defined by the resources, not in the location of the `{% resource %}` tag. Duplicate resources will not be added and any dependencies will be included as well as the resources, all in the correct order (see below for details).

Libraries can be added to CKAN from extensions using a helper function within the toolkit. See below.

In debug mode resources are served un-minified and unbundled (each resource is served separately). In non-debug mode the files are served minified and bundled (where allowed).

Important: .js and .css resources must be supplied as un-minified files. Minified files will be created. It is advised to include a .gitignore file to prevent minified files being added to the repository.

Resources within extensions

To add a resource within a extension helper function `add_resource(path, name):`

```
ckan.plugins.toolkit.add_resource('path/to/my/fanstatic/library/dir',  
    'my_fanstatic_library')
```

The first argument, `path`, is the path to the resource directory relative to the file calling the function. The second argument, `name` is the name of the library (to be used by templates when they want to include a resource from the library using the `{% resource %}` tag as shown above).

Resources will be created for the library for any .js and .css files found in the directory or it's subfolders. The resource name being the name of the file including any path needed to get to it from the resource directory. For greater control of the creation a `resource.config` file can be created and placed in the resource directory (see below for details).

resource.config

This file is used to define the resources in a directory and its sub folders. Here is an example file. The general layout of the file and allowed syntax is the same as for the .ini config file.

```
# Example resource.config file

[main]

dont_bundle = jquery.js
force_top = html5.js
order = jquery.js jed.js

[IE conditional]

lte IE 8 = html5.js block_html5_shim
IE 7 = font-awesome/css/font-awesome-ie7.css
others = html5.js

[custom render order]

block_html5_shim = 1
html5.js = 2
select2/select2.css = 9

[inline scripts]

block_html5_shim =
    var html5 = {shivMethods: false};

[depends]

vendor = jquery.js
```

[groups]

```
vendor =
  jed.js
  html5.js
  select2/select2.js
  select2/select2.css
  bootstrap/js/bootstrap-transition.js
  bootstrap/js/bootstrap-modal.js
  bootstrap/js/bootstrap-alert.js
  bootstrap/js/bootstrap-tab.js
  bootstrap/js/bootstrap-button.js
  font-awesome/css/font-awesome-ie7.css
```

[main]

This can contain the following values

force_top

The resources listed will be placed in the head of the page. This is only relevant to .js files which will by default will be added to the bottom of the page.

dont_bundle

Bundeling resources causes them to be served to the browser as a single resource to prevent multiple requests to the server. The resources listed will not be bundled. By default items will be bundled where possible. Note that .css files can only be bundled if they are in the same directory.

order

This is used to make sure that resources are created in the order specified. It should not generally be needed but is available if there are problems.

[IE conditional]

This allows IE conditionals to be wrapped around resources

```
eg <!--[if IE lte 8]><script src="my_script.js"></script><![end if]>
```

The condition is supplied followed by a list of resources that need that condition.

others

This is a special condition that means that the resource will also be available for none IE browsers.

[custom render order]

By default resources have a render order this is 10 for .css and 20 for .js resources. Sometimes we need to add resources before or after they would be included an example being the html5shim.js that needs including before .css resources. By providing a custom render order for the resource it's placement can be altered. Lower numbered resources are rendered earlier. Note that resources rendered in the head will still be placed before ones rendered in the body.

[inline scripts]

It is possible to define inline scripts in the resource.config file this can be helpful in some situations but is probably best avoided if possible.

[depends]

Some times one resource depends on another eg many scripts need jquery.js included in the page before them. External resource libraries will automatically depend on the core CKAN JavaScript modules so do not need to specify this.

[groups]

Groups of resources can be specified this allows the group to be included by just using it's name rather than having to specify each resource individuality when requesting them. The order that items are added to a group will be used to order the resources when added to the page but other factors such as dependencies, custom render order and resource type can affect the final order used.

Groups can be referred to in many places in the resource.config file eg. [depends]

Creating a new template

This is a brief tutorial covering the basics of building a common template.

Extending a base template

Firstly we need to extend a parent template to provide us with some basic page structure. This can be any other HTML page however the most common one is `page.html` which provides the full CKAN theme including header and footer.

```
{% extends "page.html" %}
```

The `page.html` template provides numerous blocks that can be extended. It's worth spending a few minutes getting familiar with what's available. The most common blocks that we'll be using are those ending with "content".

- `primary_content`: The main content area of the page.
- `secondary_content`: The secondary content (sidebar) of the page.
- `breadcrumb_content`: The contents of the breadcrumb navigation.
- `actions_content`: The content of the actions bar to the left of the breadcrumb.

Primary Content

For now we'll add some content to the main content area of the page.

```
{% block primary_content %}
  {{ super() }}

  {% block my_content %}
    <h2>{{ _('This is my content heading') }}</h2>
    <p>{{ _('This is my content') }}</p>
  {% endblock %}
{% endblock %}
```

Notice we've wrapped our own content in a block. This allows other templates to extend and possibly override this one and is extremely useful for making a them more customisable.

Secondary Content

Secondary content usually compromises of reusable modules which are pulled in as snippets. Snippets are also very useful for keeping the templates clean and allowing theme extensions to override them.

```
{% block primary_content %}
    {{ super() }}

    {% block my_sidebar_module %}
        {% snippet "snippets/my-sidebar-module.html" %}
    {% endblock %}
{% endblock %}
```

Breadcrumb and Actions

There is a consistent breadcrumb running through all the pages and often it is useful to provide additional actions that a related to the page.

```
{% block breadcrumb_content %}
    <li class="active">{% link_for _('Viewing Dataset'), controller='package', action='read', id=pkg.id %}>
{% endblock %}

{% block actions_content %}
    {{ super() }}
    <li class="active">{% link_for _('New Dataset'), controller='package', action='new', class_='btn', %}>
{% endblock %}
```

Scripts and Stylesheets

Currently scripts and stylesheets can be added by extending the `styles` and `scripts` blocks. This is soon to be replaced with the `{% resource %}` tag which manages script loading for us.

```
{% block styles %}
    {{ super() }}
    <link rel="stylesheet" href="{% url_for_static "my-style.css" %}" />
{% endblock %}

{% block scripts %}
    {{ super() }}
    <script src="{% url_for_static "my-script.js" %}"></script>
{% endblock %}
```

Summary

And that's about all there is to it be sure to check out `base.html` and `page.html` to see all the tags available for extension.

Template Blocks

These blocks can be extended by child templates to replace or extend common CKAN functionality.

Usage

There are currently two base templates *base.html* which provides the bare HTML structure such as title, head and body tags as well as hooks for adding links, stylesheets and scripts. *page.html* defines the content structure and is the template that you'll likely want to use.

To extend a template simply create a new template file and call `{% extend %}` then define the blocks that you wish to override.

Blocks in page.html

page.html extends the “page” block in *base.html* and provides the basic page structure for primary and secondary content.

header

Override the header on a page by page basis by extending this block. If making site wide header changes it is preferable to override the *header.html* file:

```
{% block header %}
    {% include "custom_header.html" %}
{% endblock %}
```

content

The content block allows you to replace the entire content section of the page with your own markup if needed:

```
{% block content %}
    <div class="custom-content">
        {% block custom_block %}{% endblock %}
    </div>
{% endblock %}
```

toolbar

The toolbar is for content to be added at the top of the page such as the breadcrumb navigation. You can remove/replace this by extending this block:

```
{# Remove the toolbar from this page. #}
{% block toolbar %}{% endblock %}
```

breadcrumb

Add a breadcrumb to the page by extending this element:

```
{% block breadcrumb %}
    {% include "breadcrumb.html" %}
{% endblock %}
```


actions

Add actions to the page by extending this element:

```
{% block actions %}
  <a class="btn" href="{{ save_url }}">Save</a>
{% endblock %}
```

primary

This block can be used to remove the entire primary content element:

```
{% block primary %}{% endblock %}
```

primary_content

The `primary_content` block can be used to add content to the page. This is the main block that is likely to be used within a template:

```
{% block primary_content %}
  <h1>My page content</h1>
  <p>Some content for the page</p>
{% endblock %}
```

secondary

This block can be used to remove the entire secondary content element:

```
{% block secondary %}{% endblock %}
```

secondary_content

The `secondary_content` block can be used to add content to the sidebar of the page. This is the main block that is likely to be used within a template:

```
{% block secondary_content %}
  <h2>A sidebar item</h2>
  <p>Some content for the item</p>
{% endblock %}
```

footer

Override the footer on a page by page basis by extending this block:

```
{% block footer %}
  {% include "custom_footer.html" %}
{% endblock %}
```

If making site wide header changes it is preferable to override the *footer.html*. Adding scripts should use the “scripts” block instead.

Blocks in base.html

doctype

Allows the DOCTYPE to be set on a page by page basis:

```
{% block doctype %}<!DOCTYPE html>{% endblock %}
```

htmltag

Allows custom attributes to be added to the <html> tag:

```
{% block htmltag %}<html lang="en-gb" class="no-js">{% endblock %}
```

headtag

Allows custom attributes to be added to the <head> tag:

```
{% block headtag %}<head data-tag="No idea what you'd add here">{% endblock %}
```

bodytag

Allows custom attributes to be added to the <body> tag:

```
{% block bodytag %}<body class="full-page">{% endblock %}
```

meta

Add custom meta tags to the page. Call `super()` to get the default tags such as charset, viewport and generator:

```
{% block meta %}
  {{ super() }}
  <meta name="author" value="Joe Bloggs" />
  <meta name="description" value="My website description" />
{% endblock %}
```

title

Add a custom title to the page by extending the title block. Call `super()` to get the default page title:

```
{% block title %}My Subtitle - {{ super() }}{% endblock %}
```

links

The links block allows you to add additional content before the stylesheets such as rss feeds and favicons in the same way as the meta block:

```
{% block link %}
  <meta rel="shortcut icon" href="custom_icon.png" />
{% endblock %}
```

styles

The styles block allows you to add additional stylesheets to the page in the same way as the meta block. Use “`super()`” to include the default stylesheets before or after your own:

```
{% block styles %}
  {{ super() }}
  <link rel="stylesheet" href="/base/css/custom.css" />
{% endblock %}
```

page

The page block allows you to add content to the page. Most of the time it is recommended that you extend one of the `page.html` templates in order to get the site header and footer. If you need a clean page then this is the block to use:

```
{% block page %}
  <div>Some other page content</div>
{% endblock %}
```

scripts

The scripts block allows you to add additional scripts to the page. Use the `super()` function to load the default scripts before/after your own:

```
{% block script %}
  {{ super() }}
  <script src="/base/js/custom.js"></script>
{% endblock %}
```

Building a JavaScript Module

CKAN makes heavy use of modules to add additional functionality to the page. Essentially all a module consists of is an object with an `.initialize()` and `.teardown()` method.

Here we will go through the basic functionality of building a simple module that sends a “favourite” request to the server when the user clicks a button.

HTML

The idea behind modules is that the element should already be in the document when the page loads. For example our favourite button will work just fine without our module JavaScript loaded.

```
<form action="/favourite" method="post" data-module="favorite">
  <button class="btn" name="package" value="101">Submit</button>
</form>
```

Here it's the `data-module="favorite"` that tells the CKAN module loader to create a new instance for this element.

JavaScript

Modules reside in the `javascript/modules` directory and should share the same name as the module. We use hyphens to delimit spaces in both filenames and modules.

```
/javascript/modules/favorite.js
```

A module can be created by calling `ckan.module()`:

```
ckan.module('favorite', function (jQuery, _) {  
    return {};  
});
```

We pass in the module name and a factory function that should return our module object. This factory gets passed a local `jQuery` object and a translation object.

Note: In order to include a module for page render inclusion within an extension it is recommended that you use `{% resource %}` within your templates. See the Resource Documentation

Initialisation

Once `ckan` has found an element on the page it creates a new instance of your module and if present calls the `.initialize()` method.

```
ckan.module('favorite', function (jQuery, _) {  
    return {  
        initialize: function () {  
            console.log('I've been called for element: %o', this.el);  
        }  
    };  
});
```

Here we can set up event listeners and other setup functions.

```
initialize: function () {  
    // Grab our button and assign it to a property of our module.  
    this.button = this.$('button');  
  
    // Watch for our favourite button to be clicked.  
    this.button.on('submit', jQuery.proxy(this._onClick, this));  
},  
_onClick: function (event) {}
```

Event Handling

Now we create our click handler for the button:

```
_onClick: function (event) {  
    event.preventDefault();  
    this.favorite();  
}
```

And this calls a `.favorite()` method. It's generally best not to do too much in event handlers it means that you can't use the same functionality elsewhere.

```
favorite: function () {  
    // The client on the sandbox should always be used to talk to the api.  
    this.sandbox.client.favoriteDataset(this.button.val());  
}
```

Notifications and Internationalisation

This submits the dataset to the API but ideally we want to tell the user what we're doing.

```
options: {
  i18n: {
    loading: _('Favouriting dataset'),
    done: _('Favourited dataset %(id)s')
  }
},
favorite: function () {
  // i18n gets a translation key from the options object.
  this.button.text(this.i18n('loading'));

  // The client on the sandbox should always be used to talk to the api.
  var request = this.sandbox.client.favoriteDataset(this.button.val())
  request.done(jQuery.proxy(this._onSuccess, this));
},
_onSuccess: function () {
  // We can perform interpolation on messages.
  var message = this.i18n('done', {id: this.button.val()});

  // Notify allows global messages to be displayed to the user.
  this.sandbox.notify(message, 'success');
}
```

Options

Displaying an id to the user isn't very friendly. We can use the `data-module` attributes to pass options through to the module.

```
<form action="/favourite" method="post" data-module="favorite" data-module-dataset="my dataset">
```

This will override the defaults in the options object.

```
ckan.module('favorite', function (jQuery, _) {
  return {
    options: {
      dataset: '',
      i18n: {...}
    }
    initialize: function () {
      console.log('this dataset is: %s', this.options.dataset);
      //=> "this dataset is: my dataset"
    }
  };
});
```

Error handling

When ever we make an Ajax request we want to make sure that we notify the user if the request fails. Again we can use `this.sandbox.notify()` to do this.

```
favorite: function () {
  // The client on the sandbox should always be used to talk to the api.
  var request = this.sandbox.client.favoriteDataset(this.button.val())
```

```
request.done(jQuery.proxy(this._onSuccess, this));
request.fail(jQuery.proxy(this._onError, this));
},
_onError: function () {
  var message = this.i18n('error', {id: this.button.val()});

  // Notify allows global messages to be displayed to the user.
  this.sandbox.notify(message, 'error');
}
```

Module Scope

You may have noticed we keep making calls to `jQuery.proxy()` within these methods. This is to ensure that `this` when the callback is called is the module it belongs to.

We have a shortcut method called `jQuery.proxyAll()` that can be used in the `.initialize()` method to do all the binding at once. It can accept method names or simply a regexp.

```
initialize: function () {
  jQuery.proxyAll(this, '_onSuccess');

  // Same as:
  this._onSuccess = jQuery.proxy(this, '_onSuccess');

  // Even better do all methods starting with _on at once.
  jQuery.proxyAll(this, /_on/);
}
```

Publish/Subscribe

Sometimes we want modules to be able to talk to each other in order to keep the page state up to date. The sandbox has the `.publish()` and `.subscribe()` methods for just this cause.

For example say we had a counter up in the header that showed how many favourite datasets the user had. This would be incorrect when the user clicked the ajax button. We can publish an event when the favorite button is successful.

```
_onSuccess: function () {
  // Notify allows global messages to be displayed to the user.
  this.sandbox.notify(message, 'success');

  // Tell other modules about this event.
  this.sandbox.publish('favorite', this.button.val());
}
```

Now in our other module ‘user-favorite-counter’ we can listen for this.

```
ckan.module('user-favorite-counter', function (jQuery, _) {
  return {
    initialize: function () {
      jQuery.proxyAll(this, /_on/);
      this.sandbox.subscribe('favorite', this._onFavorite);
    },
    teardown: function () {
      // We must always unsubscribe on teardown to prevent memory leaks.
      this.sandbox.unsubscribe('favorite', this._onFavorite);
    },
  },
});
```

```

    incrementCounter: function () {
      var count = this.el.text() + 1;
      this.el.text(count);
    },
    _onFavorite: function (id) {
      this.incrementCounter();
    }
  };
});

```

Unit Tests

Every module has unit tests. These use Mocha, Chai and Sinon to assert the expected functionality of the module.

See also:

String internationalization How to mark strings for translation.

Install frontend dependencies

The front end stylesheets are written using LESS (this depends on `node.js` being installed on the system)

Instructions for installing node can be found on the [node.js website](#). On Ubuntu 12.04 to 13.04, node.js (and npm node.js's package manager) need to be installed via a PPA:

```

$ sudo apt-add-repository ppa:chris-lea/node.js
$ sudo apt-get update

```

Now run the command to install nodejs from the repository:

```
$ sudo apt-get install nodejs
```

On Ubuntu versions later than 13.04, npm can be installed directly from Ubuntu packages

```
:: $ sudo apt-get install npm
```

For more information, refer to the [Node wiki](#).

LESS can then be installed via the node package manager which is bundled with node (or installed with apt as it is not bundled with node.js on Ubuntu). We also use `nodewatch` to make our LESS compiler a watcher style script.

cd into the `pyenv/src/ckan` and run:

```
$ npm install less@1.7.5 nodewatch
```

File structure

All front-end files to be served via a web server are located in the `public` directory (in the case of the new CKAN base theme it's `public/base`).

```

css/
  main.css
less/
  main.less
  ckan.less
  ...
javascript/

```

```
main.js
utils.js
components/
...
vendor/
  jquery.js
  jquery.plugin.js
  underscore.js
  bootstrap.css
  ...
test/
  index.html
  spec/
    main.spec.js
    utils.spec.js
  vendor/
    mocha.js
    mocha.css
    chai.js
  ...
```

All files and directories should be lowercase with hyphens used to separate words.

css Should contain any site specific CSS files including compiled production builds generated by LESS.

less Should contain all the less files for the site. Additional vendor styles should be added to the *vendor* directory and included in *main.less*.

javascript Should contain all website files. These can be structured appropriately. It is recommended that *main.js* be used as the bootstrap filename that sets up the page.

vendor Should contain all external dependencies. These should not contain version numbers in the filename. This information should be available in the header comment of the file. Library plugins should be prefixed with the library name. If a dependency has many files (such as bootstrap) then the entire directory should be included as distributed by the maintainer.

test Contains the test runner *index.html*. *vendor* contains all test dependencies and libraries. *spec* contains the actual test files. Each test file should be the filename with *.spec* appended.

Stylesheets

Because all the stylesheets are using LESS we need to compile them before beginning development. In production CKAN will look for the *main.css* file which is included in the repository. In development CKAN looks for the file *main.debug.css* which you will need to generate by running:

```
$ ./bin/less
```

This will watch for changes to all of the less files and automatically rebuild the CSS for you. To quit the script press `ctrl-c`. There is also `--production` flag for compiling the production *main.css*.

There are many LESS files which attempt to group the styles in useful groups. The main two are:

main.less: This contains *all* the styles for the website including dependencies and local styles. The only files that are excluded here are those that are conditionally loaded such as IE only CSS and large external apps (like recline) that only appear on a single page.

ckan.less: This includes all the local ckan stylesheets.

Note: Whenever a CSS change effects *main.less* it's important than after the merge into master that a \$

```
./bin/less --production should be run and committed.
```

There is a basic pattern primer available at: <http://localhost:5000/testing/primer/> that shows all the main page elements that make up the CKAN core interface.

JavaScript

The core of the CKAN JavaScript is split up into three areas.

- Core (such as i18n, pub/sub and API clients)
- *Building a JavaScript Module* (small HTML components or widgets)
- jQuery Plugins (very small reusable components)

Core

Everything in the CKAN application lives on the `ckan` namespace. Currently there are four main components that make up the core.

- Modules
- Publisher/Subscriber
- Client
- i18n/Jed

Modules

Modules are the core of the CKAN website, every component that is interactive on the page should be a module. These are then initialized by including a `data-module` attribute on an element on the page. For example:

```
:: <select name="format" data-module="autocomplete"></select>
```

The idea is to create small isolated components that can easily be tested. They should ideally not use any global objects, all functionality should be provided to them via a “sandbox” object.

There is a global factory that can be used to create new modules and jQuery and Localisation methods are available via `this.sandbox.jQuery` and `this.sandbox.translate()` respectively. To save typing these two common objects we can take advantage of JavaScript closures and use an alternative module syntax that accepts a factory function.

```
ckan.module('my-module', function (jQuery, _) {
  return {
    initialize: function () {
      // Called when a module is created.
      // jQuery and translate are available here.
    },
    teardown: function () {
      // Called before a module is removed from the page.
    }
  }
});
```

Note: A guide on creating your own modules is located in the *Building a JavaScript Module* guide.

Publisher/subscriber

There is a simple pub/sub module included under `ckan.pubsub` it's methods are available to modules via `this.sandbox.publish/subscribe/unsubscribe`. This can be used to publish messages between modules.

Modules should use the publish/subscribe methods to talk to each other and allow different areas of the UI to update where relevant.

```
ckan.module('language-picker', function (jQuery, _) {
    return {
        initialize: function () {
            var sandbox = this.sandbox;
            this.el.on('change', function () {
                sandbox.publish('change:lang', this.selected);
            });
        }
    }
});

ckan.module('language-notifier', function (jQuery, _) {
    return {
        initialize: function () {
            this.sandbox.subscribe('change:lang', function (lang) {
                alert('language is now ' + lang);
            });
        }
    }
});
```

Client

Ideally no module should use `jQuery.ajax()` to make XHR requests to the CKAN API, all functionality should be provided via the client object.

```
ckan.module('my-module', function (jQuery, _) {
    return {
        initialize: function () {
            this.sandbox.client.getCompletions(this.options.completionsUrl);
        }
    }
});
```

i18n/Jed

Jed is a Gettext implementation in JavaScript. It is used throughout the application to create translatable strings. An instance of Jed is available on the `ckan.i18n` object.

Modules get access to the `translate()` function via both the initial factory function and the `this.sandbox.translate()` object.

String interpolation can be provided using the [sprintf formatting](#). We always use the named arguments to keep in line with the Python translations. And we name the translate function passed into `ckan.module()` `_`.

```
ckan.module('my-module', function (jQuery, _) {
    return {
        initialize: function () {
```

```

// Through sandbox translation
this.sandbox.translate('my string');

// Keyword arguments
_('Hello %(name)s').fetch({name: 'Bill'}); // Hello Bill

// Multiple.
_('I like your %(color)s %(fruit)s.').fetch({color: 'red', fruit: 'apple'});

// Plurals.
_('I have %(num)d apple.')
  .ifPlural(2, "I have %(num)d apples.")
  .fetch({num: 2, fruit: 'apple'});
}
};
});

```

Life cycle

CKAN modules are initialised on dom ready. The `ckan.module.initialize()` will look for all elements on the page with a `data-module` attribute and attempt to create an instance.

```
<select name="format" data-module="autocomplete" data-module-key="id"></select>
```

The module will be created with the element, any options object extracted from `data-module-*` attributes and a new sandbox instance.

Once created the modules `initialize()` method will be called allowing the module to set themselves up.

Modules should also provide a `teardown()` method this isn't used at the moment except in the unit tests to restore state but may become useful in the future.

Internationalization

All strings within modules should be internationalized. Strings can be set in the `options.i18n` object and there is a `.i18n()` helper for retrieving them.

```

ckan.module('language-picker', function (jQuery, _) {
  return {
    options: {
      i18n: {
        hello_1: _('Hello'),
        hello_2: _('Hello %(name)s'),
        apples: function (params) {
          var n = params.num;
          return _('I have %(num)d apple').isPlural(n, 'I have %(num)d apples');
        }
      }
    },
    initialize: function () {
      // Standard example
      this.i18n('hello_1'); // "Hello"

      // String interpolation example
      var name = 'Dave';
      this.i18n('hello_2', {name: name}); // "Hello Dave"
    }
  };
});

```

```
// Plural example
var total = 1;
this.i18n('apples', {num: total}); // "I have 1 apple"
this.i18n('apples', {num: 3});    // "I have 3 apples"
}
}
});
```

jQuery plugins

Any functionality that is not directly related to ckan should be packaged up in a jQuery plug-in if possible. This keeps the modules containing only ckan specific code and allows plug-ins to be reused on other sites.

Examples of these are `jQuery.fn.slug()`, `jQuery.fn.slugPreview()` and `jQuery.proxyAll()`.

Unit tests

There is currently a test suite available at: <http://localhost:5000/base/test/index.html>

Every core component, module and plugin should have a set of unit tests. Tests can be filtered using the `grep={regex}` query string parameter.

The libraries used for the tests are as follows.

- **Mocha**: A test runner using a BDD style syntax.
- **Chai**: An assertion library (we use the assert style).
- **Sinon**: A stubbing library, can stub objects, timers and ajax requests.

Each file has a description block for it's top level object and then within that a nested description for each method that is to be tested:

```
describe('ckan.module.MyModule()', function () {
  describe('.initialize()', function () {
    it('should do something...', function () {
      // assertions.
    });
  });
});

describe('.myMethod(arg1, arg2, arg3)', function () {
});
});
```

The ``.beforeEach()`` and ``.afterEach()`` callbacks can be used to setup objects for testing (all blocks share the same scope so test variables can be attached):

```
describe('ckan.module.MyModule()', function () {
  // Pull the class out of the registry.
  var MyModule = ckan.module.registry['my-module'];

  beforeEach(function () {
    // Create a test element.
    this.el = jQuery('<div />');

    // Create a test sandbox.
    this.sandbox = ckan.sandbox();
  });
});
```

```

    // Create a test module.
    this.module = new MyModule(this.el, {}, this.sandbox);
  });

  afterEach(function () {
    // Clean up.
    this.module.teardown();
  });
});

```

Templates can also be loaded using the `.loadFixtures()` method that is available in all test contexts. Tests can be made asynchronous by setting a `done` argument in the callback (Mocha checks the arity of the functions):

```

describe('ckan.module.MyModule()', function () {

  before(function (done) {
    // Load the template once.
    this.loadFixture('my-template.html', function (html) {
      this.template = html;
      done();
    });
  });

  beforeEach(function () {
    // Assign the template to the module each time.
    this.el = this.fixture.html(this.template).children();
  });
});

```

Database migrations

When changes are made to the model classes in `ckan.model` that alter CKAN's database schema, a migration script has to be added to migrate old CKAN databases to the new database schema when they upgrade their copies of CKAN. These migration scripts are kept in `ckan.migration.versions`.

When you upgrade a CKAN instance, as part of the upgrade process you run any necessary migration scripts with the `paster db upgrade` command:

```
paster --plugin=ckan db upgrade --config={.ini file}
```

A migration script should be checked into CKAN at the same time as the model changes it is related to. Before pushing the changes, ensure the tests pass when running against the migrated model, which requires the `--ckan-migration` setting.

To create a new migration script, create a python file in `ckan/migration/versions/` and name it with a prefix numbered one higher than the previous one and some words describing the change.

You need to use the special engine provided by the SQLAlchemy Migrate. Here is the standard header for your migrate script:

```

from sqlalchemy import *
from migrate import *

```

The migration operations go in the upgrade function:

```

def upgrade(migrate_engine):
    metadata = MetaData()
    metadata.bind = migrate_engine

```

The following process should be followed when doing a migration. This process is here to make the process easier and to validate if any mistakes have been made:

1. Get a dump of the database schema before you add your new migrate scripts.

```
paster --plugin=ckan db clean --config={.ini file}
paster --plugin=ckan db upgrade --config={.ini file}
pg_dump -h host -s -f old.sql dbname
```

2. Get a dump of the database as you have specified it in the model.

```
paster --plugin=ckan db clean --config={.ini file}

#this makes the database as defined in the model
paster --plugin=ckan db create-from-model -config={.ini file}
pg_dump -h host -s -f new.sql dbname
```

3. Get apgdiff (apt-get it). It produces sql it thinks that you need to run on the database in order to get it to the updated schema.

```
apgdiff old.sql new.sql > upgrade.diff
```

(or if you don't want to install java use http://apgdiff.startnet.biz/diff_online.php)

4. The upgrade.diff file created will have all the changes needed in sql. Delete the drop index lines as they are not created in the model.
5. Put the resulting sql in your migrate script, e.g.

```
migrate_engine.execute(''update table .....; update table ....'')
```

6. Do a dump again, then a diff again to see if the the only thing left are drop index statements.
7. run nosetests with --ckan-migration flag.

It's that simple. Well almost.

- If you are doing any table/field renaming adding that to your new migrate script first and use this as a base for your diff (i.e add a migrate script with these renaming before 1). This way the resulting sql won't try to drop and recreate the field/table!
- It sometimes drops the foreign key constraints in the wrong order causing an error so you may need to rearrange the order in the resulting upgrade.diff.
- If you need to do any data transfer in the migrations then do it between the dropping of the constraints and adding of new ones.
- May need to add some tests if you are doing data migrations.

An example of a script doing it this way is `034_resource_group_table.py`. This script copies the definitions of the original tables in order to do the renaming the tables/fields.

In order to do some basic data migration testing extra assertions should be added to the migration script. Examples of this can also be found in `034_resource_group_table.py` for example.

This statement is run at the top of the migration script to get the count of rows:

```
package_count = migrate_engine.execute(''select count(*) from package'').first()[0]
```

And the following is run after to make sure that row count is the same:

```
resource_group_after = migrate_engine.execute(''select count(*) from resource_group'').first()[0]
assert resource_group_after == package_count
```

Upgrading CKAN's dependencies

The Python modules that CKAN depends on are pinned to specific versions, so we can guarantee that whenever anyone installs CKAN, they'll always get the same versions of the Python modules in their virtual environment.

Our dependencies are defined in three files:

requirements.in This file is only used to create a new version of the `requirements.txt` file when upgrading the dependencies. Contains our direct dependencies only (not dependencies of dependencies) with loosely defined versions. For example, `python-dateutil>=1.5.0,<2.0.0`.

requirements.txt This is the file that people actually use to install CKAN's dependencies into their virtualenvs. It contains every dependency, including dependencies of dependencies, each pinned to a specific version. For example, `simplejson==3.3.1`.

dev-requirements.txt Contains those dependencies only needed by developers, not needed for production sites. These are pinned to a specific version. For example, `factory-boy==2.1.1`.

We haven't created a `dev-requirements.in` file because we have too few dev dependencies, we don't update them often, and none of them have a known incompatible version.

Steps to upgrade

These steps will upgrade all of CKAN's dependencies to the latest versions that work with CKAN:

1. Create a new virtualenv: `virtualenv --no-site-packages upgrading`
2. Install the requirements with unpinned versions: `pip install -r requirements.in`
3. Save the new dependencies versions: `pip freeze > requirements.txt`. We have to do this before installing the other dependencies so we get only what was in `requirements.in`
4. Install CKAN: `python setup.py develop`
5. Install the development dependencies: `pip install -r dev-requirements.txt`
6. Run the tests to make sure everything still works (see [Testing CKAN](#)).
 - If not, try to fix the problem. If it's too complicated, pinpoint which dependency's version broke our tests, find an older version that still works, and add it to `requirements.in` (i.e., if `python-dateutil 2.0.0` broke CKAN, you'd add `python-dateutil>=1.5.0,<2.0.0`). Go back to step 1.
7. Navigate a bit on CKAN to make sure the tests didn't miss anything. Review the dependencies changes and their changelogs. If everything seems fine, go ahead and make a pull request (see `:doc'/contributing/pull-requests'`).

Doing a CKAN release

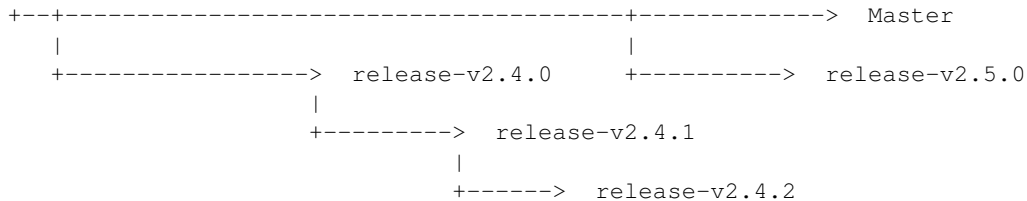
This section documents the steps followed by the development team to do a new CKAN release.

See also:

[Upgrading CKAN](#) An overview of the different kinds of CKAN release, and the process for upgrading a CKAN site to a new version.

Process overview

The process of a new release starts with the creation of a new release branch. A release branch is the one that will be stabilized and eventually become the actual released version. Release branches are always named `release-vM.m.p`, after the *major, minor and patch versions* they include. Major and minor versions are always branched from master. Patch releases are always branched from the most recent tip of the previous patch release branch.



Once a release branch has been created there is generally a three-four week period until the actual release. During this period the branch is tested and fixes cherry-picked. The whole process is described in the following sections.

Doing a beta release

Beta releases are branched off a certain point in master and will eventually become stable releases.

1. Create a new release branch:

```
git checkout -b release-v2.5.0
```

Update `ckan/__init__.py` to change the version number to the new version with a *b* after it, e.g. `2.5.0b`. Commit the change and push the new branch to GitHub:

```
git commit -am "Update version number"
git push origin release-v2.5.0
```

You will probably need to update the same file on master to increase the version number, in this case ending with an *a* (for alpha).

2. Once the release branch is created, send an announcement email with an initial call for translations, warning that at this point strings can still change, but hopefully not too much.
3. During the beta process, all changes to the release branch must be cherry-picked from master (or merged from special branches based on the release branch if the original branch was not compatible).
4. As in the master branch, if some commits involving CSS changes are cherry-picked from master, the less compiling command needs to be run on the release branch. This will update the `main.css` file:

```
./bin/less --production
git commit -am "Rebuild CSS"
git push
```

There will be a final front-end build before the actual release.

5. The beta staging site (<http://beta.ckan.org>, currently on s084) must be set to track the latest beta release branch to allow user testing. This site is updated nightly.
6. Once a week create a deb package with the latest release branch, using `betaX` iterations. Deb packages are built using [Ansible](#) scripts located at the following repo:

<https://github.com/ckan/ckan-packaging>

The repository contains further instructions on how to run the scripts, but essentially you will need access to the packaging server, and then run something like:


```
ansible-playbook package.yml -u your_user -s
```

You will be prompted for the CKAN version to package (eg 2.4.0), the iteration (eg beta1) and whether to package the DataPusher (always do it on release packages).

Packages are created by default on the */build* folder of the publicly accessible directory of the packaging server.

7. Once the translation freeze is in place (ie no changes to the translatable strings are allowed), strings need to be extracted and uploaded to [Transifex](#):

- (a) Install the Babel and Transifex libraries if necessary:

```
pip install --upgrade Babel
pip install transifex-client
```

- (b) Create a `~/.transifexrc` file if necessary with your login details (token should be left blank):

```
[https://www.transifex.com]
hostname = https://www.transifex.com
username = <username>
password = <password>
token =
```

- (c) Extract new strings from the CKAN source code into the `ckan.pot` file. The pot file is a text file that contains the original, untranslated strings extracted from the CKAN source code.:

```
python setup.py extract_messages
```

The po files are text files, one for each language CKAN is translated to, that contain the translated strings next to the originals. Translators edit the po files (on Transifex) to update the translations. We never edit the po files locally.

- (d) Run our script that checks for mistakes in the `ckan.po` files:

```
pip install polib
paster check-po-files ckan/i18n/*/LC_MESSAGES/ckan.po
```

If the script finds any mistakes correct them on Transifex and then run the `tx pull` command again, don't edit the files directly. Repeat until the script finds no mistakes.

- (e) Edit `.tx/config`, on line 4 to set the Transifex 'resource' to the new major release name (if different), using dashes instead of dots. For instance v2.4.0, v2.4.1 and v2.4.2 all share: `[ckan.2-4]`.

- (f) Update the `ckan.po` files with the new strings from the `ckan.pot` file:

```
python setup.py update_catalog --no-fuzzy-matching
```

Any new or updated strings from the CKAN source code will get into the po files, and any strings in the po files that no longer exist in the source code will be deleted (along with their translations).

We use the `--no-fuzzy-matching` option because fuzzy matching often causes problems with Babel and Transifex.

- (g) Create a new resource in the CKAN project on Transifex by pushing the new pot and po files:

```
tx push --source --translations --force
```

Because it reads the new version number in the `.tx/config` file, tx will create a new resource on Transifex rather than updating an existing resource (updating an existing resource, especially with the `--force` option, can result in translations being deleted from Transifex).

- (h) Update the `ckan.mo` files by compiling the po files:

```
python setup.py compile_catalog
```

The mo files are the files that CKAN actually reads when displaying strings to the user.

- (i) Commit all the above changes to git and push them to GitHub:

```
git commit -am "Update strings files before CKAN X.Y call for translations"
git push
```

- (j) Announce that strings for the new release are ready for translators. Send an email to the mailing lists, tweet or post it on the blog. Make sure to post a link to the correct Transifex resource (like [this one](#)) and tell users that they can register on Transifex to contribute.

- (k) A week before the translations will be closed send a reminder email.

- (l) Once the translations are closed, pull the updated strings from Transifex, check them, compile and push as described in the previous steps:

```
tx pull --all --force
paster check-po-files ckan/i18n/*/LC_MESSAGES/ckan.po
python setup.py compile_catalog
git commit -am " Update translations from Transifex"
git push
```

8. A week before the actual release, send an email to the [ckan-announce mailing list](#), so CKAN instance maintainers can be aware of the upcoming releases. List any patch releases that will be also available. Here's an [example](#) email.

Doing a proper release

Once the release branch has been thoroughly tested and is stable we can do a release.

1. Run the most thorough tests:

```
nosetests ckan/tests --ckan --ckan-migration --with-pylons=test-core.ini
```

2. Do a final build of the front-end and commit the changes:

```
paster front-end-build
git commit -am "Rebuild front-end"
```

3. Update the CHANGELOG.txt with the new version changes:

- Add the release date next to the version number
- Add the following notices at the top of the release, reflecting whether updates in requirements, database or Solr schema are required or not:

```
Note: This version requires a requirements upgrade on source installations
Note: This version requires a database upgrade
Note: This version does not require a Solr schema upgrade
```

- Check the issue numbers on the commit messages for information about the changes. The following gist has a script that uses the GitHub API to aid in getting the merged issues between releases:

<https://gist.github.com/amercader/4ec55774b9a625e815bf>

Other helpful commands are:

```
git branch -a --merged > merged-current.txt
git branch -a --merged ckan-1.8.1 > merged-previous.txt
diff merged-previous.txt merged-current.txt

git log --no-merges release-v1.8.1..release-v2.0
git shortlog --no-merges release-v1.8.1..release-v2.0
```

4. Check that the docs compile correctly:

```
rm build/sphinx -rf
python setup.py build_sphinx
```

5. Remove the beta letter in the version number in `ckan/__init__.py` (eg 1.1b -> 1.1) and commit the change:

```
git commit -am "Update version number for release X.Y"
```

6. Tag the repository with the version number, and make sure to push it to GitHub afterwards:

```
git tag -a -m '[release]: Release tag' ckan-X.Y
git push --tags
```

7. Create the final deb package and move it to the root of the [publicly accessible folder](#) of the packaging server from the `/build` folder.

Make sure to rename it so it follows the deb packages name convention:

```
python-ckan_Major.minor_amd64.deb
```

Note that we drop any patch version or iteration from the package name.

8. Upload the release to PyPI:

```
python setup.py sdist upload
```

You will need a PyPI account with admin permissions on the ckan package, and your credentials should be defined on a `~/.pypirc` file, as described [here](#). If you make a mistake, you can always remove the release file on PyPI and re-upload it.

9. Enable the new version of the docs on Read the Docs (you will need an admin account):

- (a) Go to the [Read The Docs](#) versions page and enable the relevant release (make sure to use the tag, ie ckan-X.Y, not the branch, ie release-vX.Y).
- (b) If it is the latest stable release, set it to be the Default Version and check it is displayed on <http://docs.ckan.org>.

10. Write a [CKAN Blog post](#) and send an email to the mailing list announcing the release, including the relevant bit of changelog.

11. Cherry-pick the i18n changes from the release branch onto master.

We don't generally merge or cherry-pick release branches into master, but the files in `ckan/i18n` are an exception. These files are only ever changed on release branches following the [Doing a beta release](#) instructions above, and after a release has been finalized the changes need to be cherry-picked onto master.

To find out what i18n commits there are on the release-v* branch that are not on master, do:

```
git log master..release-v* ckan/i18n
```

Then checkout the master branch, do a `git status` and a `git pull` to make sure you have the latest commits on master and no local changes. Then use `git cherry-pick` when on the master branch to cherry-pick these commits onto master. You should not get any merge conflicts. Run the `check-po-files` command

again just to be safe, it should not report any problems. Run CKAN's tests, again just to be safe. Then do `git push origin master`.

Changelog

v2.4.9 2017-09-27

- Allow overriding email headers (#3781)
- Support non-root instances on fanstatic (#3618)
- Add missing close button on organization page (#3814)

v2.4.8 2017-08-02

- Fix in organization / group form image URL field (#3661)
- Fix activity test to use utcnow (#3644)
- Changed required permission from 'update' to 'manage_group' (#3631)
- Catch invalid sort param exception (#3630)
- Choose direction of recreated package relationship depending on its type (#3626)
- Fix render_datetime for dates before year 1900 (#3611)
- Fix KeyError in 'package_create' (#3027)
- Allow slug preview to work with autocomplete fields (#2501)
- Fix filter results button not working for organization/group (#3620)
- Allow underscores in URL slug preview on create dataset (#3612)
- Create new resource view if resource format changed (#3515)
- Fixed incorrect escaping in *mail_to*
- Autocomplete fields are more responsive - 300ms timeout instead of 1s (#3693)
- Fixed dataset count display for groups (#3711)
- Restrict access to form pages (#3684)

v2.4.7 2017-03-22

- Use fully qualified urls for reset emails (#3486)

- Fix edit_resource for resource with draft state (#3480)
- Tag fix for group/organization pages (#3460)
- Fix for package_search context (#3489)

v2.4.6 2017-02-22

- Use the url_for() helper for datapusher URLs (#2866)
- Resource creation date use datetime.utcnow() (#3447)
- Fix locale error when using fix ckan.root_path
- *render_markdown* breaks links with ampersands
- Check group name and id during package creation
- Use utcnow() on dashboard_mark_activities_old (#3373)
- Fix encoding error on DataStore exception
- Datastore doesn't add site_url to resource created via API (#3189)
- Fix memberships after user deletion (#3265)
- Remove idle database connection (#3260)
- Fix package_owner_org_update action when called via the API (#2661)

v2.4.5 2017-02-22

Cancelled release

v2.4.4 2016-11-02

- Changing your user name produces an error and logs you out (#2394)
- Fix “Load more” functionality in the dashboard (#2346)
- Fix filters not working when embedding a resource view (#2657)
- Proper sanitation of header name on SlickGrid view (#2923)
- Fix unicode error when indexing field of type JSON (#2969)
- Fix group feeds returning no datasets (#2955)
- Replace MapQuest tiles in Recline with Stamen Terrain (#3162)
- Fix bulk operations not taking effect (#3199)
- Raise validation errors on group/org_member_create (#3108)
- Incorrect warnings when ckan.views.default_views is empty (#3093)
- Don't show deleted users/datasets on member_list (#3078)

v2.4.3 2016-03-31

Bug fixes:

- Use *resource.url* as *raw_resource_url* (#2873)
- Fix `DomainObject.count()` to return count (#2919)
- Add offset param to *organization_activity* (#2640)
- Prevent unicode/ascii conversion errors in DataStore
- Fix *datastore_delete* erasing the db when filters is blank (#2885)
- Avoid *package_search* exception when using *use_default_schema* (#2848)
- *resource_edit* incorrectly setting action to new instead of edit
- Encode EXPLAIN SQL before sending to datastore
- Use *ckan.site_url* to generate urls of resources (#2592)
- Don't hide actual exception on paster commands

v2.4.2 2015-12-17

Bug fixes:

- Fix Markdown rendering issue
- Return default error page on fanstatic errors
- Prevent authentication when using API callbacks

v2.4.1 2015-09-02

Note: #2554 fixes a regression where `group_list` and `organization_list` were returning extra additional fields by default, causing performance issues. This is now fixed, so the output for these actions no longer returns `users`, `extras`, etc. Also, on the homepage template the `c.groups` and `c.group_package_stuff` context variables are no longer available.

Bug fixes:

- Fix dataset count in templates and show datasets on featured org/group (#2557)
- Fix autodetect for TSV resources (#2553)
- Improve character escaping in DataStore parameters
- Fix “paster db init” when celery is configured with a non-database backend
- Fix severe performance issues with groups and orgs listings (#2554)

v2.4.0 2015-07-22

Note: This version requires a database upgrade

Note: This version requires a Solr schema upgrade

Major:

- CKAN config can now be set from environment variables and via the API (#2429)

Minor:

- API calls now faster: `group_show`, `organization_show`, `user_show`, `package_show`, `vocabulary_show` & `tag_show` (#1886, #2206, #2207, #2376)
- Require/validate current password before allowing a password change (#1940)
- Added `organization_autocomplete` action (#2125)
- Default authorization no longer allows anyone to create datasets etc (#2164)
- `organization_list_for_user` now returns organizations in hierarchy if they exist for roles set in `ckan.auth.roles_that_cascade_to_sub_groups` (#2199)
- Improved accessibility (text based browsers) focused on the page header (#2258)
- Improved IGroupForm for better customizing groups and organization behaviour (#2354)
- Admin page can now be extended to have new tabs (#2351)

Bug fixes:

- Command line `paster user` failed for non-ascii characters (#1244)
- Memory leak fixed in datastore API (#1847)
- Modifying resource didn't update it's last updated timestamp (#1874)
- Datastore didn't update if you uploaded a new file of the same name as the existing file (#2147)
- Files with really long file were skipped by datapusher (#2057)
- Multi-lingual Solr schema is now updated so it works again (#2161)
- Resource views didn't display when embedded in another site (#2238)
- `resource_update` failed if you supplied a `revision_id` (#2340)
- Recline could not plot GeoJSON on a map (#2387)
- Dataset create form 404 error if you added a resource but left it blank (#2392)
- Editing a resource view for a file that was UTF-8 and had a BOM gave an error (#2401)
- Email invites had the email address changed to lower-case (#2415)
- Default resource views not created when using a custom dataset schema (#2421, #2482)
- If the licenses pick-list was customized to remove some, datasets with old values had them overwritten when edited (#2472)
- Recline views failed on some non-ascii characters (#2490)
- Resource proxy failed if HEAD responds with 403 (#2530)
- Resource views for non-default dataset types couldn't be created (#2532)

Changes and deprecations

- The default of allowing anyone to create datasets, groups and organizations has been changed to False. It is advised to ensure you set all of the *Authorization Settings* options explicitly in your CKAN config. (#2164)

- The `package_show` API call does not return the `tracking_summary`, keys in the dataset or resources by default any more.

Any custom templates or users of this API call that use these values will need to pass: `include_tracking=True`.

- The legacy `tests` directory has moved to `tests/legacy`, the `new_tests` directory has moved to `tests` and the `new_authz.py` module has been renamed `authz.py`. Code that imports names from the old locations will continue to work in this release but will issue a deprecation warning. (#1753)
- `group_show` and `organization_show` API calls no longer return the datasets by default (#2206)
Custom templates or users of this API call will need to pass `include_datasets=True` to include datasets in the response.
- The `vocabulary_show` and `tag_show` API calls no longer returns the `packages` key - i.e. datasets that use the vocabulary or tag. However `tag_show` now has an `include_datasets` option. (#1886)
- Config option `site_url` is now required - CKAN will not abort during start-up if it is not set. (#1976)

v2.3.5 2016-11-02

- Fix “Load more” functionality in the dashboard (#2346)
- Fix filters not working when embedding a resource view (#2657)
- Proper sanitation of header name on SlickGrid view (#2923)
- Fix unicode error when indexing field of type JSON (#2969)
- Fix group feeds returning no datasets (#2955)
- Replace MapQuest tiles in Recline with Stamen Terrain (#3162)
- Fix bulk operations not taking effect (#3199)
- Raise validation errors on group/org_member_create (#3108)
- Incorrect warnings when `ckan.views.default_views` is empty (#3093)
- Don’t show deleted users/datasets on member_list (#3078)

v2.3.4 2016-03-31

Bug fixes:

- Use `resource.url` as `raw_resource_url` (#2873)
- Fix `DomainObject.count()` to return count (#2919)
- Prevent unicode/ascii conversion errors in `DataStore`
- Fix `datastore_delete` erasing the db when filters is blank (#2885)
- Avoid `package_search` exception when using `use_default_schema` (#2848)
- `resource_edit` incorrectly setting action to new instead of edit
- Use `ckan.site_url` to generate urls of resources (#2592)
- Don’t hide actual exception on paster commands

v2.3.3 2015-12-17

Bug fixes:

- Fix Markdown rendering issue
- Return default error page on fanstatic errors
- Prevent authentication when using API callbacks

v2.3.2 2015-09-02

Bug fixes: * Fix autodetect for TSV resources (#2553) * Improve character escaping in DataStore parameters * Fix “paster db init” when celery is configured with a non-database backend

v2.3.1 2015-07-22

Bug fixes:

- Resource views won’t display when embedded in another site (#2238)
- `resource_update` failed if you supplied a `revision_id` (#2340)
- Recline could not plot GeoJSON on a map (#2387)
- Dataset create form 404 error if you added a resource but left it blank (#2392)
- Editing a resource view for a file that was UTF-8 and had a BOM gave an error (#2401)
- Email invites had the email address changed to lower-case (#2415)
- Default resource views not created when using a custom dataset schema (#2421, #2482)
- If the licenses pick-list was customized to remove some, datasets with old values had them overwritten when edited (#2472)
- Recline views failed on some non-ascii characters (#2490)
- Resource views for non-default dataset types couldn’t be created (#2532)

v2.3 2015-03-04

Note: This version requires a requirements upgrade on source installations

Note: This version requires a database upgrade

Note: This version requires a Solr schema upgrade

Note: This version requires a DataPusher upgrade on source installations. You should target Data-Pusher=>0.0.6 and upgrade its dependencies.

Major:

- Completely refactored resource data visualizations, allowing multiple persistent views of the same data an interface to manage and configure them. (#1251, #1851, #1852, #2204, #2205) Check the updated documentation to know more, and the “Changes and deprecations” section for migration details:

<http://docs.ckan.org/en/latest/maintaining/data-viewer.html>

- Responsive design for the default theme, that allows nicer rendering across different devices (#1935)
- Improved DataStore filtering and full text search capabilities (#1792, #1830, #1838, #1815)
- Added new extension points to modify the DataStore behaviour (#1725)
- Simplified two-step dataset creation process (#1659)
- Ability for users to regenerate their own API keys (#1412)
- New `package_patch` action to allow individual fields dataset updates (#1416, #1679)
- Changes on the authentication mechanism to allow more secure setups (`httponly` and `secure` cookies, disable CORS, etc). (#2004, #2050, #2052 ...) See “Changes and deprecations” section for more details and “Troubleshooting” for migration instructions.
- Better support for custom dataset types (#1795, #2083)
- Extensions can combine free-form extras and `convert_to_extras` fields (#1894)
- Updated documentation theme, now clearer and responsive (#1845)

Minor:

- Adding custom fields tutorial (#790)
- Add metadata created and modified fields to the dataset page (#655)
- Improve IFacets plugin interface docstrings (#781)
- Remove help string from API calls (#1318)
- Add “datapusher submit” command to upload existing resources data (#1792)
- More template blocks to allow for easier extension maintenance (#1301)
- CKAN API - remove help string from standard calls (#1318)
- Hide activity by selected users on activity stream (#1330)
- Documentation and clarification about “CKAN Flavored Markdown” (#1332)
- Resource formats are now guessed automatically (#1350)
- New JavaScript modules tutorial (#1377)
- Allow overriding dataset, group, org validation (#1400)
- Remove ResourceGroups, show `package_id` on resources (#1407)
- Better errors for NAVL junk (#1418)
- DataPusher integration improvements (#1446)
- Allow people to create unowned datasets when they belong to an org (#1473)
- Add `res_type` to Solr schema (#1495)
- Separate data and metadata licenses on create dataset page (#1503)
- Allow CKAN (and paster) to find config from envvar (#1597)
- Added `xlsx` and `tsv` to the defaults for `ckan.datapusher.formats`. (#1644)
- Add resource extras to Solr search index (#1709)
- Prevent packages update in `organization_update` (#1711)
- Programatically log user in after registration (#1721)
- New plugin interfaces: `IValidators.get_validators` and `IConverters.get_converters` (#1841)

- Index resource name in Solr (#1905)
- Update search index after membership changes (#1917)
- resource_show: use package_show to get validated data (#1921)
- Serve placeholder images locally (#1951)
- Don't get all datasets when loading the org in the dataset page (#1978)
- Text file preview - lack of vertical scroll bar for long files (#1982)
- Changes to allow better use of custom group types in IGroupForm extensions (#1987)
- Remove moderated edits (#2006)
- package_create: allow sysadmins to set package ids (#2102)
- Enable a logged in user to move dataset to another organization (#2218)
- Move PDF views into a separate extension (#2270)
- Do not provide email configuration in default config file (#2273)
- Add custom DataStore SQLAlchemy properties (#2279)

Bug fixes:

- Set up stats extension as namespace plugin (#291)
- Fix visibility validator for datasets (#1188)
- Select boxes with autocomplete are clearing their placeholders (#1278)
- Default search ordering on organization home page is broken (#1368)
- related_list logic function throws a 503 without any parameters (#1384)
- Exception on group dictize due to 'with_capacity' on context (#1390)
- Wrong template on Add member page (#1392)
- Overflowing email address on user page (#1398)
- The reset password e-mail is using an incorrect translation string (#1409)
- You can't view a group when there is an IGroupForm (#1420)
- Disabling activity_streams borks editing groups and user (#1421)
- Use a more secure default for the repoze secret key (#1422)
- Duplicated Required Fields notice on Group form (#1426)
- UI language reset after account creation (#1429)
- num_followers and package_count not in default_group_schema (#1434)
- Fix extras deletion (#1449)
- Fix resource reordering (#1450)
- Datastore callback fails when browser url is different from site_url (#1451)
- sysadmins should not create datasets without org when config is set (#1453)
- Member Editing Fixes (#1454)
- Bulk editing broken on IE7 (#1455)
- Fix group deletion on IE7 (#1460)

- Organization ATOM feed is broken (#1463)
- Users can not delete a dataset that not belongs to an organization (#1471)
- Error during authorization in datapusher_hook (#1487)
- Wrong datapusher hook callback URL on non-root deployments (#1490)
- Wrong breadcrumbs on new dataset form and resource pages (#1491)
- Atom feed Content-Type returned as 'text/html' (#1504)
- Invite to organization causes Internal Server error (#1505)
- Dataset tags autocomplete doesn't work (#1512)
- Activity Stream from: Organization Error group not found (#1519)
- Improve password hashing algorithm (#1530)
- Can't download resources with geojson extension (#1534)
- All datasets for featured group/organization shown on home page (#1569)
- Able to list private datasets via the API (#1580)
- Don't lowercase the names of uploaded files (#1584)
- Show more facets only if there are more facts to show (#1612)
- resource_create should break when called without URL (#1641)
- Creating a DataStore resource with the package_id fails for a normal user (#1652)
- Fix package permission checks for create+update (#1664)
- bulk_process page for non-existent organization throws Exception (#1682)
- Catch NotFound error in resource_proxy (#1684)
- Fix int_validator (#1692)
- Current date indexed on empty "_date" fields (#1701)
- Possible to show a resource inside an arbitrary dataset (#1707)
- Edit member page shows wrong fields (#1723)
- Insecure content warning when running Recline under SSL (#1729)
- Flash messages not displayed as part of page.html (#1743)
- package_show response includes solr rubbish when using ckan.cache_validated_datasets (#1764)
- "Add some resources" link shown to unauthorized users (#1766)
- email notifications via paster plugin post erroneously demands authentication (#1767)
- Inserting empty arrays in JSON type fields in datastore fails (#1776)
- Ordering a dataset listing loses the existing filters (#1791)
- Don't delete all cookies whose names start with "ckan" (#1793)
- Upgrade some major requirements (eg SQLAlchemy, Requests) (#1817, #1819)
- list of member roles disappears on add member page (#1873)
- Stats plugin should only show active datasets (#1936)
- Featured group on homepage not linking to group (#1996)

- `--reload` doesn't work on the 'paster serve' command (#2013)
- Can not override auth config options from tests (#2035)
- Fix `resource_create` authorization (#2037)
- `package_search` gives internal server error if page < 1 (#2042)
- Fix organization pagination (#2141)
- Resource extras can not be updated (#2158)
- `package_show` doesn't validate when a custom schema is used (#2175)
- Update jQuery minified version to match the unminified one (#1750)
- Fix exception during database upgrade (#2029)
- Fix resources disappearing on dataset update (#1779)
- Fix activity stream queries performance on large instances (#2008)
- Only link to http, https and ftp resource urls (#2085)
- Avoid private and deleted datasets on stats plugin (#1936)
- Fix tags count and group links in stats extension (#1649)
- Make `resource_create` auth work against `package_update` (#2037)
- Fix DataStore permissions check on startup (#1374)
- Fix datastore docs link (#2044)
- Clean up field names before rendering the Recline table (#2319)
- Don't "normalize" resource URL in recline view (#2324)
- Don't assume resource format is there on text preview (#2320)
- And many, many more!

Changes and deprecations

- By convention, view plugin names now end with `_view` rather than `_preview` (eg `recline_view` rather than `recline_preview`). You will need to update them on the [ckan.plugins](#) setting.
- The way resource visualizations are created by default has changed. You might need to set the [ckan.views.default_views](#) configuration option and run a migration command on existing instances. Please refer to the migration guide for more details:

<http://docs.ckan.org/en/latest/maintaining/data-viewer.html#migrating-from-previous-ckan-versions>

- The PDF Viewer extension has been moved to a separate extension: <https://github.com/ckan/ckanext-pdfview>. Please install it separately if you are using the `pdf_view` plugin (or the old `pdf_preview` one).
- The action API (v3) no longer returns the full help for the action on each request. It rather includes a link to a separate call to get the action help string.
- The `user_show` API call does not return the `datasets`, `num_followers` or `activity` keys by default any more.

Any custom templates or users of this API call that use these values will need to specify parameters: `include_datasets` or `include_num_followers`.

`activity` has been removed completely as it was actually a list of revisions, rather than the activity stream. If you want the actual activity stream for a user, call `user_activity_list` instead.

- The output of `resource_show` now contains a `package_id` key that links to the parent dataset.
- `helpers.get_action()` (or `h.get_action()` in templates) is deprecated.

Since action functions raise exceptions and templates cannot catch exceptions, it's not a good idea to call action functions from templates.

Instead, have your controller method call the action function and pass the result to your template using the `extra_vars` param of `render()`.

Alternatively you can wrap individual action functions in custom template helper functions that handle any exceptions appropriately, but this is likely to make your the logic in your templates more complex and templates are difficult to test and debug.

Note that `logic.get_action()` and `toolkit.get_action()` are *not* deprecated, core code and plugin code should still use `get_action()`.

- Cross-Origin Resource Sharing (CORS) support is no longer enabled by default. Previously, Access-Control-Allow-* response headers were added for all requests, with Access-Control-Allow-Origin set to the wildcard value *. To re-enable CORS, use the new `ckan.cors` configuration settings ([ckan.cors.origin_allow_all](#) and [ckan.cors.origin_whitelist](#)).
- The `HttpOnly` flag will be set on the authorization cookie by default. For enhanced security, we recommend using the `HttpOnly` flag, but this behaviour can be changed in the `Repoze.who` settings detailed in the Config File Options documentation ([who.httponly](#)).
- The OpenID login option has been removed and is no longer supported. See “Troubleshooting” if you are upgrading an existing CKAN instance as you may need to update your `who.ini` file.

Template changes

- Note to people with custom themes: If you've changed the `{% block secondary_content %}` in `templates/package/search.html` pay close attention as this pull request changes the structure of that template block a little.

Also: There's a few more bootstrap classes (especially for grid layout) that are now going to be in the templates. Take a look if any of the following changes might effect your content blocks:

<https://github.com/ckan/ckan/pull/1935>

Troubleshooting:

- Login does not work, for existing and new users.

You need to update your existing `who.ini` file.

- In the `[plugin:auth_tkt]` section, replace:

```
use = ckan.config.middleware:ckan_auth_tkt_make_app
```

with:

```
use = ckan.lib.auth_tkt:make_plugin
```

- In `[authenticators]`, add the `auth_tkt` plugin

Also see the next point for OpenID related changes.

- Exception on first load after upgrading from a previous CKAN version:

```
ImportError: <module 'ckan.lib.authenticator' from '/usr/lib/ckan/default/src/ckan/ckan/lib/auth
```

or:

```
ImportError: No module named openid
```

There are OpenID related configuration options in your `who.ini` file which are no longer supported.

This file is generally located in `/etc/ckan/default/who.ini` but its location may vary if you used a custom deployment.

The options that you need to remove are:

- The whole `[plugin:openid]` section
- In `[general]`, replace:

```
challenge_decider = repoze.who.plugins.openid.classifiers:openid_challenge_decider
```

with:

```
challenge_decider = repoze.who.classifiers:default_challenge_decider
```

- In `[identifiers]`, remove `openid`
- In `[authenticators]`, remove `ckan.lib.authenticator:OpenIDAuthenticator`
- In `[challengers]`, remove `openid`

This is a diff with the whole changes:

<https://github.com/ckan/ckan/pull/2058/files#diff-2>

Also see the previous point for other `who.ini` changes.

v2.2.4 2015-12-17

Bug fixes:

- Fix Markdown rendering issue
- Return default error page on fanstatic errors
- Prevent authentication when using API callbacks

v2.2.3 2015-07-22

Bug fixes:

- Allow uppercase emails on user invites (#2415)
- Fix broken boolean validator (#2443)
- Fix auth check in `resources_list.html` (#2037)
- Key error on resource proxy (#2425)
- Ignore `revision_id` passed to resources (#2340)
- Add reset for `reset_key` on successful password change (#2379)

v2.2.2 2015-03-04

Bug fixes:

- Update jQuery minified version to match the unminified one (#1750)
- Fix exception during database upgrade (#2029)
- Fix resources disappearing on dataset update (#1779)
- Fix activity stream queries performance on large instances (#2008)
- Only link to http, https and ftp resource urls (#2085)
- Avoid private and deleted datasets on stats plugin (#1936)
- Fix tags count and group links in stats extension (#1649)
- Make resource_create auth work against package_update (#2037)
- Fix DataStore permissions check on startup (#1374)
- Fix datastore docs link (#2044)
- Fix resource extras getting lost on resource update (#2158)
- Clean up field names before rendering the Recline table (#2319)
- Don't "normalize" resource URL in recline view (#2324)
- Don't assume resource format is there on text preview (#2320)

v2.2.1 2014-10-15

Bug fixes:

- Organization image_url is not displayed in the dataset view. (#1934)
- list of member roles disappears on add member page if you enter a user that doesn't exist (#1873)
- group/organization_member_create do not return a value. (#1878)
- i18n: Close a tag in French translation in Markdown syntax link (#1919)
- organization_list_for_user() fixes (#1918)
- Don't show private datasets to group members (#1902)
- Incorrect link in Organization snippet on dataset page (#1882)
- Prevent reading system tables on DataStore SQL search (#1871)
- Ensure that the DataStore is running on legacy mode when using PostgreSQL < 9.x (#1879)
- Select2 in the Tags field is broken (#1864)
- Edit user encoding error (#1436)
- Able to list private datasets via the API (#1580)
- Insecure content warning when running Recline under SSL (#1729)
- Add quotes to package ID in Solr query in _bulk_update_dataset to prevent Solr errors with custom dataset IDs. (#1853)
- Ordering a dataset listing loses the existing filters (#1791)

- Inserting empty arrays in JSON type fields in datastore fails (#1776)
- email notifications via paster plugin post erroneously demands authentication (#1767)
- “Add some resources” link shown to unauthorized users (#1766)
- Current date indexed on empty “*_date” fields (#1701)
- Edit member page shows wrong fields (#1723)
- programatically log user in after registration (#1721)
- Dataset tags autocomplete doesn’t work (#1512)
- Deleted Users bug (#1668)
- UX problem with previous and next during dataset creation (#1598)
- Catch NotFound error in resources page (#1685)
- _tracking page should only respond to POST (#1683)
- bulk_process page for non-existent organization throws Exception (#1682)
- Fix package permission checks for create+update (#1664)
- Creating a DataStore resource with the package_id fails for a normal user (#1652)
- Trailing whitespace in resource URLs not stripped (#1634)
- Move the closing div inside the block (#1620)
- Fix open redirect (#1419)
- Show more facets only if there are more facts to show (#1612)
- Fix breakage in package groups page (#1594)
- Fix broken links in RSS feed (#1589)
- Activity Stream from: Organization Error group not found (#1519)
- DataPusher and harvester collision (#1500)
- Can’t download resources with geojson extension (#1534)
- Oversized Forgot Password button and field (#1508)
- Invite to organization causes Internal Server error (#1505)

v2.2 2014-02-04

Note: This version does not require a requirements upgrade on source installations

Note: This version requires a database upgrade

Note: This version requires a Solr schema upgrade (The Solr schema file has been renamed, the schema file from the previous release is compatible with this version, but users are encouraged to point to the new one, see “API changes and deprecations”)

Major:

- Brand new automatic importer of tabular data to the DataStore, the DataPusher. This is much more robust and simple to deploy and maintain than its predecessor (ckanext-datastorer). Whole new UI for re-importing data to the DataStore and view the import logs (#932, #938, #940, #981, #1196, #1200 ...)

- Completely revamped file uploads that allow closer integration with resources and the DataStore, as well as making easier to integrate file uploads in other features. For example users can now upload images for organizations and groups. See “API changes and deprecations” if you are using the current FileStore. (#1273, #1173 ...)
- UI and API endpoints for resource reordering (#1277)
- Backend support for organization hierarchy, allowing parent and children organizations. Frontend needs to be implemented in extensions (#1038)
- User invitations: it is now possible to create new users with just their email address. An invite email is sent to them, allowing to change their user name and password (#1178)
- Disable user registration with a configuration option (#1226)
- Great effort in improving documentation, specially for customizing CKAN, with a complete tutorial for writing extensions and customizing the theme. User and sysadmin guides have also been moved to the main documentation (#943, #847, #1253)

Minor:

- Homepage modules to allow predefined layouts (#1126)
- Ability to delete users (#1163)
- Dedicated dataset groups page for displaying and managing them (#1102)
- Implement organization_purge and group_purge action functions (#707)
- Improve package_show performance (#1078)
- Support internationalization of rendered dates and times (#1041)
- Improve plugin load handling (#549)
- Authorization function auditing for action functions (#1060)
- Improve datetime rendering (#518)
- New SQL indexes to improve performance (#1164)
- Changes in requirements management (#1149)
- Add offset/limit to package_list action (#1179)
- Document all available configuration options (#848)
- Make CKAN sqlalchemy 0.8.4 compatible (#1427)
- UI labelling and cleanup (#1030)
- Better UX for empty groups/orgs (#1094)
- Improve performance of group_dictize when the group has a lot of packages (#1208)
- Hide __extras from extras on package_show (#1218)
- “Clear all” link within each facet block is unnecessary (#1263)
- Term translations of organizations (#1274)
- ‘-reset-db’ option for when running tests (#1304)

Bug fixes:

- Fix plugins load/unload issues (#547)
- Improve performance when new_activities not needed (#1013)

- Resource preview breaks when CSV headers include percent sign (#1067)
- Package index not rebuilt when resources deleted (#1081)
- Don't accept invalid URLs in resource proxy (#1106)
- UI language reset after account creation (#1429)
- Catch non-integer facet limits (#1118)
- Error when deleting custom tags (#1114)
- Organization images do not display on Organization user dashboard page (#1127)
- Can not reactivate a deleted dataset from the UI (#607)
- Non-existent user profile should give error (#1068)
- Recaptcha not working in CKAN 2.0 (jinja templates) (#1070)
- Groups and organizations can be visited with interchangeable URLs (#1180)
- Dataset Source (url) and Version fields missing (#1187)
- Fix problems with private / public datasets and organizations (#1188)
- group_show should never return private data (#1191)
- When editing a dataset, the organization field is not set (#1199)
- Fix resource_delete action (#1216)
- Fix trash purge action redirect broken for CKAN instances not at / (#1217)
- Title edit for existing dataset changes the URL (#1232)
- 'facet.limit' in package_search wrongly handled (#1237)
- h.SI_number_span doesn't close correctly (#1238)
- CkanVersionException wrongly raised (#1241)
- (group|organization)_member_create only accepts username (and not id) (#1243)
- package_create uses the wrong parameter for organization (#1257)
- ValueError for non-int limit and offset query params (#1258)
- Visibility field value not kept if there are errors on the form (#1265)
- package_list should not return private datasets (#1295)
- Fix 404 on organization activity stream and about page (#1298)
- Fix placeholder images broken on non-root locations (#1309)
- "Add Dataset" button shown on org pages when not authorized (#1348)
- Fix exception when visiting organization history page (#1359)
- Fix search ordering on organization home page (#1368)
- datastore_search_sql failing for some anonymous users (#1373)
- related_list logic function throws a 503 without any parameters (#1384)
- Disabling activity_streams borks editing groups and user (#1421)
- Member Editing Fixes (#1454)
- Bulk editing broken in IE7 (#1455)

- Fix group deletion in IE7 (#1460)
- And many, many more!

API changes and deprecations:

- The Solr schema file is now always named `schema.xml` regardless of the CKAN version. Old schema files have been kept for backwards compatibility but users are encouraged to point to the new unified one (#1314)
- The FileStore and file uploads have been completely refactored and simplified to only support local storage backend. The links from previous versions of the FileStore to hosted files will still work, but there is a command available to migrate the files to new Filestore. See this page for more details: <http://docs.ckan.org/en/latest/filestore.html#filestore-21-to-22-migration>
- By default, the authorization for any action defined from an extension will require a logged in user, otherwise a `ckan.logic.NotAuthorized` exception will be raised. If an action function allows anonymous access (eg search, show status, etc) the `auth_allow_anonymous_access` decorator (available on the plugins toolkit) must be used (#1210)
- `package_search` now returns results with custom schemas applied like `package_show`, a `use_default_schema` parameter was added to request the old behaviour, this change may affect customized search result templates (#1255)
- The `ckan.api_url` configuration option has been completely removed and it can no longer be used (#960)
- The `edit` and `after_update` methods of `IPackageController` plugins are now called when updating a resource using the web frontend or the `resource_update` API action (#1052)
- Dataset moderation has been deprecated, and the code will probably be removed in later CKAN versions (#1139)
- Some front end libraries have been updated, this may affect existing custom themes: Bootstrap 2.0.3 > 2.3.2, Font Awesome 3.0.2 > 3.2.1, jQuery 1.7.2 > 1.10.2 (#1082)
- SQLite is officially no longer supported as the tests backend

Troubleshooting:

- Exception on startup after upgrading from a previous CKAN version:

```
AttributeError: 'instancemethod' object has no attribute 'auth_audit_exempt'
```

Make sure that you are not loading a 2.1-only plugin (eg `datapusher-ext`) and update all the plugin in your configuration file to the latest stable version.

- Exception on startup after upgrading from a previous CKAN version:

```
File "/usr/lib/ckan/default/src/ckan/ckan/lib/dictization/model_dictize.py", line 330, in pa
    result_dict['metadata_modified'] = pkg.metadata_modified.isoformat()
AttributeError: 'NoneType' object has no attribute 'isoformat'
```

One of the database changes on this version is the addition of a `metadata_modified` field in the package table, that was filled during the DB migration process. If you have previously migrated the database and revert to an older CKAN version the migration process may have failed at this step, leaving the fields empty. Also make sure to restart running processes like harvesters after the update to make sure they use the new code base.

v2.1.6 2015-12-17

Bug fixes:

- Fix Markdown rendering issue
- Return default error page on fanstatic errors
- Prevent authentication when using API callbacks

v2.1.5 2015-07-22

Bug fixes:

- Fix broken boolean validator (#2443)
- Key error on resource proxy (#2425)
- Ignore revision_id passed to resources (#2340)
- Add reset for reset_key on successful password change (#2379)

v2.1.4 2015-03-04

Bug fixes:

- Only link to http, https and ftp resource urls (#2085)
- Avoid private and deleted datasets on stats plugin (#1936)
- Fix tags count and group links in stats extension (#1649)
- Make resource_create auth work against package_update (#2037)
- Fix DataStore permissions check on startup (#1374)
- Fix datastore docs link (#2044)
- Fix resource extras getting lost on resource update (#2158)
- Clean up field names before rendering the Recline table (#2319)
- Don't "normalize" resource URL in recline view (#2324)
- Don't assume resource format is there on text preview (#2320)

v2.1.3 2014-10-15

Bug fixes:

- Organization image_url is not displayed in the dataset view. (#1934)
- i18n: Close a tag in French translation in Markdown syntax link (#1919)
- organization_list_for_user() fixes (#1918)
- Incorrect link in Organization snippet on dataset page (#1882)
- Prevent reading system tables on DataStore SQL search (#1871)

- Ensure that the DataStore is running on legacy mode when using PostgreSQL < 9.x (#1879)
- Edit user encoding error (#1436)
- Able to list private datasets via the API (#1580)
- Insecure content warning when running Recline under SSL (#1729)
- Add quotes to package ID in Solr query in `_bulk_update_dataset` to prevent Solr errors with custom dataset IDs. (#1853)
- Ordering a dataset listing loses the existing filters (#1791)
- Inserting empty arrays in JSON type fields in datastore fails (#1776)
- programmatically log user in after registration (#1721)
- Deleted Users bug (#1668)
- Catch NotFound error in resources page (#1685)
- `bulk_process` page for non-existent organization throws Exception (#1682)
- Default search ordering on organization home page is broken (#1368)
- Term translations of organizations (#1274)
- Preview fails on private datastore resources (#1221)
- Strip whitespace from title in model dictize (#1228)

v2.1.2 2014-02-04

Bug fixes:

- Fix context for group/about `setup_template_variables` (#1433)
- Call `setup_template_variables` in group/org read, about and `bulk_process` (#1281)
- Remove repeated sort code in `package_search` (#1461)
- Ensure that `check_access` is called on `activity_create` (#1421)
- Fix visibility validator (#1188)
- Remove `p.toolkit.auth_allow_anonymous_access` as it is not available on 2.1.x (#1373)
- Add `organization_revision_list` to avoid exception on org history page (#1359)
- Fix activity and about organization pages (#1298)
- Show 404 instead of login page on user not found (#1068)
- Don't show Add Dataset button on org pages unless authorized (#1348)
- Fix `datastore_search_sql` authorization function (#1373)
- Fix extras deletion (#1449)
- Better word breaking on long words (#1398)
- Fix activity and about organization pages (#1298)
- Remove limit of number of arguments passed to `user add` command.
- Fix `related_list` logic function (#1384)
- Avoid `UnicodeEncodeError` on feeds when params contains non ascii characters

v2.1.1 2013-11-8

Bug fixes:

- Fix errors on preview on non-root locations (#960)
- Fix place-holder images on non-root locations (#1309)
- Don't accept invalid URLs in resource proxy (#1106)
- Make sure came_from url is local (#1039)
- Fix logout redirect in non-root locations (#1025)
- Wrong auth checks for sysadmins on package_create (#1184)
- Don't return private datasets on package_list (#1295)
- Stop tracking failing when no lang/encoding headers (#1192)
- Fix for paster db clean command getting frozen
- Fix organization not set when editing a dataset (#1199)
- Fix PDF previews (#1194)
- Fix preview failing on private datastore resources (#1221)

v2.1 2013-08-13

Note: This version requires a requirements upgrade on source installations

Note: This version requires a database upgrade

Note: This version does not require a Solr schema upgrade

Note: The `json_preview` plugin has been renamed to `text_preview` (see #266). If you are upgrading CKAN from a previous version you need to change the plugin name on your CKAN config file after upgrading to avoid a `PluginNotFound` exception.

Major:

- Bulk updates of datasets within organizations (delete, make public/private) (#278)
- Organizations and Groups search (#303)
- Generic text preview extension for JSON, XML and plain text files (#226)
- Improve consistency of the Action API (#473)
- IAuthenticator interface for plugging into authorization platforms (Work in progress) (#1007)
- New clearer dashboard with more information easier to access (#626)
- New `rebuild_fast` command to speed up reindex using multiple cores (#700)
- Complete restructure of the documentation, with updated sections on installation, upgrading, release process, etc and guidelines on how to write new documentation (#769 and multiple others)

Minor:

- Add group members page to templates (#844)
- Show search facets on organization page (#776)

- Changed default sort ordering (#869)
- More consistent display of buttons across pages (#890)
- History page ported to new templates (#368)
- More blocks to templates to allow further customization (#688)
- Improve imports from lib.helpers (#262)
- Add support for callback parameter on Action API (#414)
- Create site_user at startup (#952)
- Add warning before deleting an organization (#803)
- Remove flags from language selector (#822)
- Hide the Data API button when datastore is disabled (#752)
- Pin all requirements and separate minimal requirements in a separate file (#491, #1149)
- Better preview plugin selection (#1002)
- Add new functions to the plugins toolkit (#1015)
- Improve ExampleIDatasetFormPlugin (#2750)
- Extend h.sorted_extras() to do substitutions and auto clean keys (#440)
- Separate default database for development and testing (#517)
- More descriptive Solr exceptions when indexing (#674)
- Validate datastore input through schemas (#905)

Bug fixes:

- Fix 500 on password reset (#264)
- Fix exception when indexing a wrong date on a _date field (#267)
- Fix datastore permissions issues (#652)
- Placeholder images are not linked with h.url_for_static (#948)
- Explore dropdown menu is hidden behind other resources in IE (#915)
- Buttons interrupt file uploading (#902)
- Fix resource proxy encoding errors (#896)
- Enable streaming in resource proxy (#989)
- Fix cache_dir and beaker paths on deployment.ini_tmpl (#888)
- Fix multiple issues on create dataset form on IE (#881)
- Fix internal server error when adding member (#869)
- Fix license faceting (#853)
- Fix exception in dashboard (#830)
- Fix Google Analytics integration (#827)
- Fix ValueError when resource size is not an integer (#1009)
- Catch NotFound on new resource when package does not exist (#1010)
- Fix Celery configuration to allow overriding from config (#1027)

- `came_from` after login is validated to not redirect to another site (#1039)
- And many, many more!

Deprecated and removed:

- The `json_preview` plugin has been replaced by a new `text_preview` one. Please update your config files if using it. (#226)

Known issues:

- Under certain authorization setups the frontend for the groups functionality may not work as expected (See #1176 #1175).

v2.0.8 2015-12-17

Bug fixes:

- Fix Markdown rendering issue
- Return default error page on fanstatic errors
- Prevent authentication when using API callbacks

v2.0.7 2015-07-22

Bug fixes:

- Fix broken boolean validator (#2443)
- Key error on resource proxy (#2425)
- Ignore `revision_id` passed to resources (#2340)
- Add reset for `reset_key` on successful password change (#2379)

v2.0.6 2015-03-04

Bug fixes:

- Only link to http, https and ftp resource urls (#2085)
- Avoid private and deleted datasets on stats plugin (#1936)
- Fix tags count and group links in stats extension (#1649)
- Make `resource_create` auth work against `package_update` (#2037)
- Fix datastore docs link (#2044)
- Fix resource extras getting lost on resource update (#2158)
- Clean up field names before rendering the Recline table (#2319)
- Don't "normalize" resource URL in recline view (#2324)
- Don't assume resource format is there on text preview (#2320)

v2.0.5 2014-10-15

Bug fixes:

- `organization_list_for_user()` fixes (#1918)
- Incorrect link in Organization snippet on dataset page (#1882)
- Prevent reading system tables on DataStore SQL search (#1871)
- Ensure that the DataStore is running on legacy mode when using PostgreSQL < 9.x (#1879)
- Current date indexed on empty “*_date” fields (#1701)
- Able to list private datasets via the API (#1580)
- Insecure content warning when running Recline under SSL (#1729)
- Inserting empty arrays in JSON type fields in datastore fails (#1776)
- Deleted Users bug (#1668)

v2.0.4 2014-02-04

Bug fixes:

- Fix extras deletion (#1449)
- Better word breaking on long words (#1398)
- Fix activity and about organization pages (#1298)
- Show 404 instead of login page on user not found (#1068)
- Remove limit of number of arguments passed to `user add` command.
- Fix `related_list` logic function (#1384)

v2.0.3 2013-11-8

Bug fixes:

- Fix errors on preview on non-root locations (#960)
- Don't accept invalid URLs in resource proxy (#1106)
- Make sure `came_from` url is local (#1039)
- Fix logout redirect in non-root locations (#1025)
- Don't return private datasets on `package_list` (#1295)
- Stop tracking failing when no lang/encoding headers (#1192)
- Fix for `paster db clean` command getting frozen

v2.0.2 2013-08-13

Bug fixes:

- Fix markdown in group descriptions (#303)
- Fix resource proxy encoding errors (#896)
- Fix datastore exception on first run (#907)
- Enable streaming in resource proxy (#989)
- Fix in user search (#1024)
- Fix Celery configuration to allow overriding from config (#1027)
- Undefined function on organizations controller (#1036)
- Fix license not translated in orgs/groups (#1040)
- Fix link to documentation from the footer (#1062)
- Fix missing close breadcrumb tag in org templates (#1071)
- Fix recently_changed_packages_activity_stream function (#1159)
- Fix Recline map sidebar not showing in IE 7-8 (#1133)

v2.0.1 2013-06-11

Bug fixes:

- Use IDatasetForm schema for resource_update (#897)
- Fixes for CKAN being run on a non-root URL (#948, #913)
- Fix resource edit errors losing info (#580)
- Fix Czech translation (#900)
- Allow JSON filters for datastore_search on GET requests (#917)
- Install vdm from the Python Package Index (#764)
- Allow extra parameters on Solr queries (#739)
- Create site user at startup if it does not exist (#952)
- Fix modal popups positioning (#828)
- Fix wrong redirect on dataset form on IE (#963)

v2.0 2013-05-10

Note: Starting on v2.0, issue numbers with four digits refer to the old ticketing system at <http://trac.ckan.org> and the ones with three digits refer to GitHub issues. For example:

- #3020 is <http://trac.ckan.org/ticket/3020>
- #271 is <https://github.com/ckan/ckan/issues/271>

Some GitHub issues URLs will redirect to GitHub pull request pages.

Note: v2.0 is a huge release so the changes listed here are just the highlights. Bug fixes are not listed.

Note: This version requires a requirements upgrade on source installations

Note: This version requires a database upgrade

Note: This version requires a Solr schema upgrade

Organizations based authorization (see [Organizations and authorization](#)): CKAN’s new “organizations” feature replaces the old authorization system with a new one based on publisher organizations. It replaces the “Publisher Profile and Workflow” feature from CKAN 1.X, any instances relying on it will need to be updated.

- New organization-based authorization and organization of datasets
- Supports private datasets
- Publisher workflow
- New authorization ini file options

New frontend (see [Theming guide](#)): CKAN’s frontend has been completely redesigned, inside and out. There is a new default theme and the template engine has moved from Genshi to Jinja2. Any custom templates using Genshi will need to be updated, although there is a [ckan.legacy_templates](#) setting to aid in the migration.

- Block-based template inheritance
- Custom jinja tags: {`% ckan_extends %`}, {`% snippet %`} and {`% url_for %`} (#2502, #2503)
- CSS “primer” page for theme developers
- We’re now using LESS for CSS
- Scalable font icons (#2563)
- Social sharing buttons (google plus, facebook, twitter) (this replaces the ckanext-social extension)
- Three-stage dataset creation form (#2501)
- New *paster front-end-build* command does everything needed to build the frontend for a production CKAN site (runs *paster less* to compile the css files, *paster minify* to minify the css and js files, etc.)

Plugins & Extensions:

- New plugins toolkit provides a stable set of utility and helper functions for CKAN plugins to depend on.
- The IDatasetForm plugin interface has been redesigned (note: this breaks backwards-compatibility with existing IDatasetForm plugins) (#649)
- Many IDatasetForm bugs were fixed
- New example extensions in core, and better documentation for the relevant plugin interfaces: `example_ihelpers` (#447), `example_idatasetform` (#2750), hopefully more to come in 2.1!
- New IFacets interface that allows to modify the facets shown on various pages. (#400)
- The `get_action()` function now automatically adds ‘model’ and ‘session’ to the context dict (this saves on boiler-plate code, and means plugins don’t have to import `ckan.model` in order to call `get_action()`) (#172)

Activity Streams, Following & User Dashboard:

- New visual design for activity streams (#2941)
- Group activity streams now include activities for changes to any of the group’s datasets (#1664)

- Group activity streams now appear on group pages (previously they could only be retrieved via the api)
- Dataset activity streams now appear on dataset pages (previously they could only be retrieved via the api) (#3024)
- Users can now follow groups (previously you could only follow users or datasets) (#3005)
- Activity streams and following are also supported for organizations (#505)
- When you're logged into CKAN, you now get a notifications count in the top-right corner of the site, telling you how many new notifications you have on your dashboard. Clicking on the count takes you to your dashboard page to view your notifications. (#3009)
- Optionally, you can also receive notifications by email when you have new activities on your dashboard (#1635)
- Infinite scrolling of activity streams (if you scroll to the bottom of a an activity stream, CKAN will automatically load more activities) (#3018)
- Redesigned user dashboard (#3028):
 - New dropdown-menu enables you to filter you dashboard activity stream to show only activities from a particular user, dataset, group or organization that you're following
 - New sidebar shows previews and unfollow buttons (when the activity stream is filtered)
- New `ckan.activity_streams_enabled` config file setting allows you to disable the generation of activity streams (#654)

Data Preview:

- PDF files preview (#2203)
- JSON files preview
- HTML pages preview (in an iframe) (#2888)
- New plugin extension point that allows plugins to add custom data previews for different data types (#2961)
- Improved Recline Data Explorer previews (CSV files, Excel files..)
- Plain text files preview

API:

- The Action API is now CKAN's default API, and the API documentation has been rewritten (#357)

Other highlights:

- CKAN now has continuous integration testing at <https://travis-ci.org/ckan/ckan/>
- Dataset pages now have `<link rel="alternate" type="application/rdf+xml">` links in the HTML headers, allows linked-data tools to find CKAN's RDF rendering of a dataset's metadata (#413)
- CKAN's DataStore and Data API have been rewritten, and now use PostgreSQL instead of elasticsearch, so there's no need to install elasticsearch anymore (this feature was also back-ported to CKAN 1.8) (#2733)
- New Config page for sysadmins (`/ckan-admin/config`) enables sysadmins to set the site title, tag line, logo, the intro text shown on the front page, the about text shown on the `/about` page, select a theme, and add custom CSS (#2302, #2781)
- New `paster color` command for creating color schemes
- Fanstatic integration (#2371):
 - CKAN now uses Fanstatic to specify required static resource files (js, css..) for web pages
 - Enables each page to only include the static files that it needs, reducing page loads

- Enables CKAN to use bundled and minified static files, further reducing page loads
- CKAN's new *paster minify* command is used to create minified js and css files (#2950) (also see *paster front-end-build*)
- CKAN will now recognise common file format strings such as “application/json”, “JSON”, “.json” and “json” as a single file type “json” (#2416)
- CKAN now supports internalization of strings in javascript files, the new *paster trans* command is used to pull translatable strings out of javascript files (#2774, #2750)
- `convert_to/from_extras` have been fixed to not add quotes around strings (#2930)
- Updated CKAN coding standards (#3020) and CONTRIBUTING.rst file
- Built-in page view counting and ‘popular’ badges on datasets and resources There’s also a paster command to export the tracking data to a csv file (#195)
- Updated CKAN Coding Standards and new CONTRIBUTING.rst file
- You can now change the sort ordering of datasets on the dataset search page

Deprecated and removed:

- The IGenshiStreamFilter plugin interface is deprecated (#271), use the ITemplateHelpers plugin interface instead
- The Model, Search and Util APIs are deprecated, use the Action API instead
- Removed `restrict_template_vars` config setting (#2257)
- Removed deprecated `facet_title()` template helper function, use `get_facet_title()` instead (#2257)
- Removed deprecated `am_authorized()` template helper function, use `check_access()` instead (#2257)
- Removed deprecated `datetime_to_datestr()` template helper function (#2257)

v1.8.2 2013-08-13

Bug fixes:

- Fix for using harvesters with organization setup
- Refactor for user update logic
- Tweak resources visibility query

v1.8.1 2013-05-10

Bug fixes:

- Fixed possible XSS vulnerability on html input (#703)
- Fix unicode template 500 error (#808)
- Fix error on related controller

v1.8 2012-10-19

Note: This version requires a requirements upgrade on source installations

Note: This version requires a database upgrade

Note: This version does not require a Solr schema upgrade

Major

- New ‘follow’ feature that allows logged in users to follow other users or datasets (#2304)
- New user dashboard that shows an activity stream of all the datasets and users you are following. Thanks to Sven R. Kunze for his work on this (#2305)
- New version of the Datastore. It has been completely rewritten to use PostgreSQL as backend, it is more stable and fast and supports SQL queries (#2733)
- Clean up and simplifying of CKAN’s dependencies and source install instructions. Ubuntu 12.04 is now supported for source installs (#2428,#2592)
- Big speed improvements when indexing datasets (#2788)
- New action API reference docs, which individually document each function and its arguments and return values (#2345)
- Updated translations, added Japanese and Korean translations

Minor

- Add source install upgrade docs (#2757)
- Mark more strings for translation (#2770)
- Allow sort ordering of dataset listings on group pages (#2842)
- Reenable simple search option (#2844)
- Editing organization removes all datasets (#2845)
- Accessibility enhancements on templates

Bug fixes

- Fix for relative url being used when doing file upload to local storage
- Various fixes on IGroupFrom (#2750)
- Fix group dataset sort (#2722)
- Fix adding existing datasets to organizations (#2843)
- Fix 500 error in related controller (#2856)
- Fix for non-open licenses appearing open
- Editing organization removes all datasets (#2845)

API changes and deprecation:

- Template helper functions are now restricted by default. By default only those helper functions listed in `lib.helpers.__allowed_functions__` are available to templates. The full functions can still be made available by setting `ckan.restrict_template_vars = false` in your ini file. Only restricted functions will be allowed in future versions of CKAN.

- Deprecated functions related to the old faceting data structure have been removed: *helpers.py:facet_items()*, *facets.html:facet_sidebar()*, *facets.html:facet_list_items()*. Internal use of the old facets datastructure (attached to the context, *c.facets*) has been superseded by use of the improved facet data structure, *c.search_facets*. The old data structure is still available on *c.facets*, but is deprecated, and will be removed in future versions. (#2313)

v1.7.4 2013-08-13

Bug fixes:

- Refactor for user update logic
- Tweak resources visibility query

v1.7.3 2013-05-10

Bug fixes:

- Fixed possible XSS vulnerability on html input (#703)

v1.7.2 2012-10-19

Minor:

- Documentation enhancements regarding file uploads

Bug fixes:

- Fixes for lincences i18n
- Remove sensitive data from user dict (#2784)
- Fix bug in feeds controller (#2869)
- Show dataset author and maintainer names even if they have no emails
- Fix URLs for some Amazon buckets
- Other minor fixes

v1.7.1 2012-06-20

Minor:

- Documentation enhancements regarding install and extensions (#2505)
- Home page and search results speed improvements (#2402,#2403)
- I18n: Added Greek translation and updated other ones (#2506)

Bug fixes:

- UI fixes (#2507)
- Fixes for i18n login and logout issues (#2497)

- Date on add/edit resource breaks if offset is specified (#2383)
- Fix in organizations read page (#2509)
- Add synchronous_search plugin to deployment.ini template (#2521)
- Inconsistent language on license dropdown (#2575)
- Fix bug in translating lists in multilingual plugin
- Group autocomplete doesn't work with multiple words (#2373)
- Other minor fixes

v1.7 2012-05-09

Major:

- Updated SOLR schema (#2327). Note: This will require an update of the SOLR schema file and a reindex.
- Support for Organization based workflow, with membership determining access permissions to datasets (#1669,#2255)
- Related items such as visualizations, applications or ideas can now be added to datasets (#2204)
- Restricted vocabularies for tags, allowing grouping related tags together (#1698)
- Internal analytics that track number of views and downloads for datasets and resources (#2251)
- Consolidated multilingual features in an included extension (#1821,#1820)
- Atom feeds for publishers, tags and search results (#1593,#2277)
- RDF dump paster command (#2303)
- Better integration with the DataStore, based on ElasticSearch, with nice helper docs (#1797)
- Updated the Recline data viewer with new features such as better graphs and a map view (#2236,#2283)
- Improved and redesigned documentation (#2226,#2245,#2248)

Minor:

- Groups can have an image associated (#2275)
- Basic resource validation (#1711)
- Ability to search without accents for accented words (#906)
- Weight queries so that title is more important than rest of body (#1826)
- Enhancements in the dataset and resource forms (#1506)
- OpenID can now be disabled (#1830)
- API and forms use same validation (#1792)
- More robust bulk search indexing, with options to ignore exceptions and just refresh (#1616i,#2232)
- Modify where the language code is placed in URLs (#2261)
- Simplified licenses list (#1359)
- Add extension point for dataset view (#1741)

Bug fixes:

- Catch exceptions on the QA archiver (#1809)
- Error when changing language when CKAN is mounted in URL (#1804)
- Naming of a new package/group can clash with a route (#1742)
- Can't delete all of a package's resources over REST API (#2266)
- Group edit form didn't allow adding multiple datasets at once (#2292)
- Fix layout bugs in IE 7 (#1788)
- Bug with Portuguese translation and Javascript (#2318)
- Fix broken parse_rfc_2822 helper function (#2314)

v1.6 2012-02-24

Major:

- Resources now have their own pages, as well as showing in the Dataset (#1445, #1449)
- Group pages enhanced, including in-group search (#1521)
- User pages enhanced with lists of datasets (#1396) and recent activity (#1515)
- Dataset view page decluttered (#1450)
- Tags not restricted to just letters and dashes (#1453)
- Stats Extension and Storage Extension moved into core CKAN (#1576, #1608)
- Ability to mounting CKAN at a sub-URL (#1401, #1659)
- 5 Stars of Openness ratings show by resources, if ckanext-qa is installed (#1583)
- Recline Data Explorer (for previewing and plotting data) improved and v2 moved into core CKAN (#1602, #1630)

Minor:

- 'About' page rewritten and easily customisable in the config (#1626)
- Gravatar picture displayed next to My Account link (#1528)
- 'Delete' button for datasets (#1425)
- Relationships API more RESTful, validated and documented (#1695)
- User name displayed when logged in (#1529)
- Database dumps now exclude deleted packages (#1623)
- Dataset/Tag name length now limited to 100 characters in API (#1473)
- 'Status' API call now includes installed extensions (#1488)
- Command-line interface for list/read/deleting datasets (#1499)
- Slug API calls tidied up and documented (#1500)
- Users nagged to add email address if missing from their account (#1413)
- Model and API for Users to become Members of a Group in a certain Capacity (#1531, #1477)
- Extension interface to adjust search queries, indexing and results (#1547, #1738)
- API for changing permissions (#1688)

Bug fixes:

- Group deletion didn't work (#1536)
- metadata_created used to return an entirely wrong date (#1546)
- Unicode characters in field-specific API search queries caused exception (since CKAN 1.5) (#1798)
- Sometimes task_status errors weren't being recorded (#1483)
- Registering or Logging in failed silently when already logged in (#1799)
- Deleted packages were browseable by administrators and appeared in dumps (#1283, #1623)
- Facicon was a broken link unless corrected in config file (#1627)
- Dataset search showed last result of each page out of order (#1683)
- 'Simple search' mode showed 0 packages on home page (#1709)
- Occasionally, 'My Account' shows when user is not logged in (#1513)
- Could not change language when on a tag page that had accented characters or dataset creation page (#1783, #1791)
- Editing package via API deleted its relationships (#1786)

v1.5.1 2012-01-04

Major:

- Background tasks (#1363, #1371, #1408)
- Fix for security issue affecting CKAN v1.5 (#1585)

Minor:

- Language support now excellent for European languages: en de fr it es no sv pl ru pt cs sr ca
- **Web UI improvements:**
 - Resource editing refreshed
 - Group editing refreshed
 - Indication that group creation requires logging-in (#1004)
 - Users' pictures displayed using Gravatar (#1409)
 - 'Welcome' banner shown to new users (#1378)
 - Group package list now ordered alphabetically (#1502)
- Allow managing a dataset's groups also via package entity API (#1381)
- Dataset listings in API standardised (#1490)
- Search ordering by modification and creation date (#191)
- Account creation disallowed with Open ID (create account in CKAN first) (#1386)
- User name can be modified (#1386)
- Email address required for registration (for password reset) (#1319)
- Atom feeds hidden for now
- New config options to ease CSS insertion into the template (#1380)

- Removed ETag browser cache headers (#1422)
- CKAN version number and admin contact in new 'status_show' API (#1087)
- Upgrade SQLAlchemy to 0.7.3 (compatible with Postgres up to 9.1) (#1433)
- SOLR schema is now versioned (#1498)

Bug fixes:

- Group ordering on main page was alphabetical but should be by size (since 1.5) (#1487)
- Package could get added multiple times to same Group, distorting Group size (#1484)
- Search index corruption when multiple CKAN instances on a server all storing the same object (#1430)
- Dataset property metadata_created had wrong value (since v1.3) (#1546)
- Tag browsing showed tags for deleted datasets (#920)
- User name change field validation error (#1470)
- You couldn't edit a user with a unicode email address (#1479)
- Package search API results missed the extra fields (#1455)
- OpenID registration disablement explained better (#1532)
- Data upload (with ckanext-storage) failed if spaces in the filename (#1518)
- Resource download count fixed (integration with ckanext-googleanalytics) (#1451)
- Multiple CKANs with same dataset IDs on the same SOLR core would conflict (#1462)

v1.5 2011-11-07

Deprecated due to security issue #1585**Major:**

- **New visual theme (#1108)**
 - Package & Resource edit overhaul (#1294/#1348/#1351/#1368/#1296)
 - JS and CSS reorganization (#1282, #1349, #1380)
- Apache Solr used for search in core instead of Postgres (#1275, #1361, #1365)
- Authorization system now embedded in the logic layer (#1253)
- Captcha added for user registration (#1307, #1431)
- UI language translations refreshed (#1292, #1350, #1418)
- Action API improved with docs now (#1315, #1302, #1371)

Minor:

- Cross-Origin Resource Sharing (CORS) support (#1271)
- Strings to translate into other languages tidied up (#1249)
- Resource format autocomplete (#816)
- Database disconnection gives better error message (#1290)
- Log-in cookie is preserved between sessions (#78)

- Extensions can access formalchemy forms (#1301)
- ‘Dataset’ is the new name for ‘Package’ (#1293)
- Resource standard fields added: type, format, size (#1324)
- Listing users speeded up (#1268)
- Basic data preview functionality moved to core from QA extension (#1357)
- Admin Extension merged into core CKAN (#1264)
- URLs in the Notes field are automatically linked (#1320)
- Disallow OpenID for account creation (but can be linked to accounts) (#1386)
- Tag name now validated for max length (#1418)

Bug fixes:

- Purging of revisions didn’t work (since 1.4.3) (#1258)
- Search indexing wasn’t working for SOLR (since 1.4.3) (#1256)
- Configuration errors were being ignored (since always) (#1172)
- Flash messages were temporarily held-back when using proxy cache (since 1.3.2) (#1321)
- On login, user told ‘welcome back’ even if he’s just registered (#1194)
- Various minor exceptions cropped up (mostly since 1.4.3) (#1334, #1346)
- Extra field couldn’t be set to original value when key deleted (#1356)
- JSONP callback parameter didn’t work for the Action API (since 1.4.3) (#1437)
- The same tag could be added to a package multiple times (#1331)

v1.4.3.1 2011-09-30

Minor:

- Added files to allow debian packaging of CKAN
- Added Catalan translation

Bug fixes:

- Incorrect Group creation form parameter caused exception (#1347)
- Incorrect AuthGroup creation form parameter caused exception (#1346)

v1.4.3 2011-09-13

Major:

- Action API (API v3) (beta version) provides powerful RPC-style API to CKAN data (#1335)
- Documentation overhaul (#1142, #1192)

Minor:

- Viewing of a package at a given date (as well as revision) with improved UI (#1236)
- Extensions can now add functions to the logic layer (#1211)

- Refactor all remaining database code out of the controllers and into the logic layer (#1229)
- Any OpenID log-in errors that occur are now displayed (#1228)
- ‘url’ field added to search index (e9214)
- Speed up tag reading (98d72)
- Cope with new WebOb version 1 (#1267)
- Avoid exceptions caused by bots hitting error page directly (#1176)
- Too minor to mention: #1234,

Bug fixes:

- Re-adding tags to a package failed (since 1.4.1 in Web UI, 1.4 in API) (#1239)
- Modified revisions retrieved over API caused exception (since 1.4.2) (#1310)
- Whichever language you changed to, it announced “Language set to: English” (since 1.3.1) (#1082)
- Incompatibilities with Python 2.5 (since 1.3.4.1 and maybe earlier) (#1325)
- You could create an authorization group without a name, causing exceptions displaying it (#1323)
- Revision list wasn’t showing deleted packages (b21f4)
- User editing error conditions handled badly (#1265)

v1.4.2 2011-08-05

Major:

- Packages revisions can be marked as ‘moderated’ (#1141, #1147)
- Password reset facility (#1186/#1198)

Minor:

- Viewing of a package at any revision (#1236)
- API POSTs can be of Content-Type “application/json” as alternative to existing “application/x-www-form-urlencoded” (#1206)
- Caching of static files (#1223)

Bug fixes:

- When you removed last row of resource table, you couldn’t add it again - since 1.0 (#1215)
- Adding a tag to package that had it previously didn’t work - since 1.4.1 in UI and 1.4.0 in API (#1239)
- Search index was not updated if you added a package to a group - since 1.1 (#1140)
- Exception if you had any Groups and migrated between CKAN v1.0.2 to v1.2 (migration 29) - since v1.0.2 (#1205)
- API Package edit requests returned the Package in a different format to usual - since 1.4 (#1214)
- API error responses were not all JSON format and didn’t have correct Content-Type (#1214)
- API package delete doesn’t require a Content-Length header (#1214)

v1.4.1 2011-06-27

Major:

- Refactor Web interface to use logic layer rather than model objects directly. Forms now defined in navl schema and designed in HTML template. Forms use of Formalchemy is deprecated. (#1078)

Minor:

- Links in user-supplied text made less attractive to spammers (nofollow) #1181
- Package change notifications - remove duplicates (#1149)
- Metadata dump linked to (#1169)
- Refactor authorization code to be common across Package, Group and Authorization Group (#1074)

Bug fixes

- Duplicate authorization roles were difficult to delete (#1083)

v1.4 2011-05-19

Major:

- Authorization forms now in grid format (#1074)
- Links to RDF, N3 and Turtle metadata formats provided by semantic.ckan.net (#1088)
- Refactor internal logic to all use packages in one format - a dictionary (#1046)
- A new button for administrators to change revisions to/from a deleted state (#1076)

Minor:

- Etags caching can now be disabled in config (#840)
- Command-line tool to check search index covers all packages (#1073)
- Command-line tool to load/dump postgres database (#1067)

Bug fixes:

- Visitor can't create packages on new CKAN install - since v1.3.3 (#1090)
- OpenID user pages couldn't be accessed - since v1.3.2 (#1056)
- Default site_url configured to ckan.net, so pages obtains CSS from ckan.net- since v1.3 (#1085)

v1.3.3 2011-04-08

Major:

- Authorization checks added to editing Groups and PackageRelationships (#1052)
- API: Added package revision history (#1012, #1071)

Minor:

- API can take auth credentials from cookie (#1001)
- Theming: Ability to set custom favicon (#1051)

- Importer code moved out into ckanext-importlib repo (#1042)
- API: Group can be referred to by ID (in addition to name) (#1045)
- Command line tool: rights listing can now be filtered (#1072)

Bug fixes:

- SITE_READ role setting couldn't be overridden by sysadmins (#1044)
- Default 'reader' role too permissive (#1066)
- Resource ordering went wrong when editing and adding at same time (#1054)
- GET followed by PUTting a package stored an incorrect license value (#662)
- Sibling package relationships were shown for deleted packages (#664)
- Tags were displayed when they only apply to deleted packages (#920)
- API: 'Last modified' time was localised - now UTC (#1068)

v1.3.2 2011-03-15

Major:

- User list in the Web interface (#1010)
- CKAN packaged as .deb for install on Ubuntu
- Resources can have extra fields (although not in web interface yet) (#826)
- CSW Harvesting - numerous of fixes & improvements. Ready for deployment. (#738 etc)
- Language switcher (82002)

Minor:

- Wordpress integration refactored as a Middleware plugin (#1013)
- Unauthorized actions lead to a flash message (#366)
- Resources Groups to group Resources in Packages (#956)
- Plugin interface for authorization (#1011)
- Database migrations tested better and corrected (#805, #998)
- Government form moved out into ckanext-dgu repo (#1018)
- Command-line user authorization tools extended (#1038, #1026)
- Default user roles read from config file (#1039)

Bug fixes:

- Mounting of filesystem (affected versions since 1.0.1) (#1040)
- Resubmitting a package via the API (affected versions since 0.6?) (#662)
- Open redirect (affected v1.3) (#1026)

v1.3 2011-02-18

<http://ckan.org/milestone/ckan-v1.3>

Highlights of changes:

- **Package edit form improved:**
 - field instructions (#679)
 - name autofilled from title (#778)
- Group-based access control - Authorization Groups (#647)
- Metadata harvest job management (#739, #884, #771)
- CSW harvesting now uses owslib (#885)
- Package creation authorization is configurable (#648)
- Read-only maintenance mode (#777)
- Stats page (#832) and importer (#950) moved out into CKAN extensions

Minor:

- site_title and site_description config variables (#974)
- Package creation/edit timestamps (#806)
- Caching configuration centralised (#828)
- Command-line tools - sysadmin management (#782)
- Group now versioned (#231)

v1.2 2010-11-25

<http://ckan.org/milestone/ckan-v1.2>

Highlights of changes:

- Package edit form: attach package to groups (#652) & revealable help
- Form API - Package/Harvester Create/New (#545)
- Authorization extended: user groups (#647) and creation of packages (#648)
- Plug-in interface classes (#741)
- WordPress twentyten compatible theming (#797)
- Caching support (ETag) (#693)
- Harvesting GEMINI2 metadata records from OGC CSW servers (#566)

Minor:

- New API key header (#466)
- Group metadata now revisioned (#231)

v1.1 2010-08-10

<http://ckan.org/milestone/v1.1>

Highlights of changes:

- Changes to the database cause notifications via AMQP for clients (#325)
- Pluggable search engines (#317), including SOLR (#353)
- API is versioned and packages & groups can be referred to by invariant ID (#313)
- Resource search in API (#336)
- Visual theming of CKAN now easy (#340, #320)
- Greater integration with external Web UIs (#335, #347, #348)
- Plug-ins can be configured to handle web requests from specified URIs and insert HTML into pages.

Minor:

- Search engine optimisations e.g. alphabetical browsing (#350)
- CSV and JSON dumps improved (#315)

v1.0.2 2010-08-27

- Bugfix: API returns error when creating package (#432)

v1.0.1 2010-06-23

Functionality:

- API: Revision search 'since id' and revision model in API
- API: Basic API versioning - packages specified by ID (#313)
- Pluggable search - initial hooks
- Customisable templates (#340) and external UI hooks (#335)

Bugfixes:

- Revision primary key lost in migrating data (#311)
- Local authority license correction in migration (#319)
- I18n formatting issues
- Web i/f searches foreign characters (#319)
- Data importer timezone issue (#330)

v1.0 2010-05-11

CKAN comes of age, having been used successfully in several deployments around the world. 56 tickets covered in this release. See: <http://ckan.org/milestone/v1.0>

Highlights of changes:

- Package edit form: new pluggable architecture for custom forms (#281, #286)
- Package revisions: diffs now include tag, license and resource changes (#303)
- Web interface: visual overhaul (#182, #206, #214-#227, #260) including a tag cloud (#89)
- i18n: completion in Web UI - now covers package edit form (#248)
- API extended: revisions (#251, #265), feeds per package (#266)
- Developer documentation expanded (#289, #290)
- Performance improved and CKAN stress-tested (#201)
- Package relationships (Read-Write in API, Read-Only in Web UI) (#253-257)
- Statistics page (#184)
- Group edit: add multiple packages at once (#295)
- Package view: RDF and JSON formatted metadata linked to from package page (#247)

Bugfixes:

- Resources were losing their history (#292)
- Extra fields now work with spaces in the name (#278, #280) and international characters (#288)
- Updating resources in the REST API (#293)

Infrastructural:

- Licenses: now uses external License Service ('licenses' Python module)
- Changesets introduced to support distributed revisioning of CKAN data - see doc/distributed.rst for more information.

v0.11 2010-01-25

Our biggest release so far (55 tickets) with lots of new features and improvements. This release also saw a major new production deployment with the CKAN software powering <http://data.gov.uk/> which had its public launch on Jan 21st!

For a full listing of tickets see: <<http://ckan.org/milestone/v0.11>>. Main highlights:

- Package Resource object (multiple download urls per package): each package can have multiple 'resources' (urls) with each resource having additional metadata such as format, description and hash (#88, #89, #229)
- "Full-text" searching of packages (#187)
- Semantic web integration: RDFization of all data plus integration with an online RDF store (e.g. for <http://www.ckan.net/> at <http://semantic.ckan.net/> or Talis store) (#90 #163)
- Package ratings (#77 #194)
- i18n: we now have translations into German and French with deployments at <http://de.ckan.net/> and <http://fr.ckan.net/> (#202)
- Package diffs available in package history (#173)
- Minor:
 - Package undelete (#21, #126)
 - Automated CKAN deployment via Fabric (#213)

- Listings are sorted alphabetically (#195)
- Add extras to rest api and to ckanclient (#158 #166)
- Infrastructural:
 - Change to UUIDs for revisions and all domain objects
 - Improved search performance and better pagination
 - Significantly improved performance in API and WUI via judicious caching

v0.10 2009-09-30

- Switch to repoze.who for authentication (#64)
- Explicit User object and improved user account UI with recent edits etc (#111, #66, #67)
- Generic Attributes for Packages (#43)
- Use sqlalchemy-migrate to handle db/model upgrades (#94)
- “Groups” of packages (#105, #110, #130, #121, #123, #131)
- Package search in the REST API (#108)
- Full role-based access control for Packages and Groups (#93, #116, #114, #115, #117, #122, #120)
- New CKAN logo (#72)
- Infrastructural:
 - Upgrade to Pylons 0.9.7 (#71)
 - Convert to use formalchemy for all forms (#76)
 - Use paginate in webhelpers (#118)
- Minor:
 - Add author and maintainer attributes to package (#91)
 - Change package state in the WUI (delete and undelete) (#126)
 - Ensure non-active packages don’t show up (#119)
 - Change tags to contain any character (other than space) (#62)
 - Add Is It Open links to package pages (#74)

v0.9 2009-07-31

- (DM!) Add version attribute for package
- Fix purge to use new version of vdm (0.4)
- Link to changed packages when listing revision
- Show most recently registered or updated packages on front page
- Bookmarklet to enable easy package registration on CKAN
- Usability improvements (package search and creation on front page)
- Use external list of licenses from license repository

- Convert from py.test to nosetests

v0.8 2009-04-10

- View information about package history (ticket:53)
- Basic datapkg integration (ticket:57)
- Show information about package openness using icons (ticket:56)
- One-stage package create/registration (r437)
- Reinstate package attribute validation (r437)
- Upgrade to vdm 0.4

v0.7 2008-10-31

- Convert to use SQLAlchemy and vdm v0.3 (v. major)
- Atom/RSS feed for Recent Changes
- Package search via name and title
- Tag lists show number of associated packages

v0.6 2008-07-08

- Autocompletion (+ suggestion) of tags when adding tags to a package.
- Paginated lists for packages, tags, and revisions.
- RESTful machine API for package access, update, listing and creation.
- API Keys for users who wish to modify information via the REST API.
- Update to vdm v0.2 (SQLObject) which fixes ordering of lists.
- Better immunity to SQL injection attacks.

v0.5 2008-01-22

- Purging of a Revision and associated changes from cli and wui (ticket:37)
- Make data available in machine-usable form via sql dump (ticket:38)
- Upgrade to Pylons 0.9.6.* and deploy (ticket:41)
- List and search tags (ticket:33)
- (bugfix) Manage reserved html characters in urls (ticket:40)
- New spam management utilities including (partial) blacklist support

v0.4 2007-07-04

- Preview support when editing a package (ticket:36).
- Correctly list IP address of not logged in users (ticket:35).
- Improve read action for revision to list details of changed items (r179).
- Sort out deployment using modpython.

v0.3 2007-04-12

- System now in a suitable state for production deployment as a beta
- Domain model versioning via the vdm package (currently released separately)
- Basic Recent Changes listing log messages
- User authentication (login/logout) via open ID
- License page
- Myriad of small fixes and improvements

v0.2 2007-02

- Complete rewrite of ckan to use pylons web framework
- Support for full CRUD on packages and tags
- No support for users (authentication)
- No versioning of domain model objects

v0.1 2006-05

NB: not an official release

- Almost functional system with support for persons, packages
- Tag support only half-functional (tags are per package not global)
- Limited release and file support

C

- `ckan.lib.helpers`, 265
- `ckan.logic.action.create`, 148
- `ckan.logic.action.delete`, 162
- `ckan.logic.action.get`, 131
- `ckan.logic.action.patch`, 161
- `ckan.logic.action.update`, 156
- `ckan.logic.converters`, 218
- `ckan.logic.validators`, 215
- `ckan.plugins.interfaces`, 193
- `ckan.tests.controllers`, 332
- `ckan.tests.factories`, 324
- `ckan.tests.helpers`, 325
- `ckan.tests.lib`, 333
- `ckan.tests.logic.action`, 329
- `ckan.tests.logic.auth`, 330
- `ckan.tests.logic.test_schema`, 332
- `ckan.tests.migration`, 334
- `ckan.tests.model`, 333
- `ckan.tests.plugins`, 333
- `ckanext.datastore.logic.action`, 71

Symbols

`_()` (built-in function), 263

A

`abort()` (ckan.plugins.interfaces.IAuthenticator method), 209

`actions` (built-in class), 264

`activity_create()` (in module ckan.logic.action.create), 154

`activity_detail_list()` (in module ckan.logic.action.get), 143

`activity_div()` (in module ckan.lib.helpers), 269

`activity_type_exists()` (in module ckan.logic.validators), 216

`add_url_param()` (in module ckan.lib.helpers), 270

`after_begin()` (ckan.plugins.interfaces.ISession method), 194

`after_commit()` (ckan.plugins.interfaces.ISession method), 195

`after_create()` (ckan.plugins.interfaces.IPackageController method), 198

`after_create()` (ckan.plugins.interfaces.IResourceController method), 199

`after_delete()` (ckan.plugins.interfaces.IMapper method), 194

`after_delete()` (ckan.plugins.interfaces.IPackageController method), 198

`after_delete()` (ckan.plugins.interfaces.IResourceController method), 200

`after_flush()` (ckan.plugins.interfaces.ISession method), 194

`after_insert()` (ckan.plugins.interfaces.IMapper method), 194

`after_load()` (ckan.plugins.interfaces.IPluginObserver method), 200

`after_map()` (ckan.plugins.interfaces.IRoutes method), 194

`after_rollback()` (ckan.plugins.interfaces.ISession method), 195

`after_search()` (ckan.plugins.interfaces.IPackageController method), 198

`after_show()` (ckan.plugins.interfaces.IPackageController method), 198

`after_unload()` (ckan.plugins.interfaces.IPluginObserver method), 200

`after_update()` (ckan.plugins.interfaces.IMapper method), 194

`after_update()` (ckan.plugins.interfaces.IPackageController method), 198

`after_update()` (ckan.plugins.interfaces.IResourceController method), 199

`am_following_dataset()` (in module ckan.logic.action.get), 145

`am_following_group()` (in module ckan.logic.action.get), 145

`am_following_user()` (in module ckan.logic.action.get), 145

`app_globals` (built-in variable), 263

`are_there_flash_messages()` (in module ckan.lib.helpers), 266

`auto_log_message()` (in module ckan.lib.helpers), 269

B

`before_commit()` (ckan.plugins.interfaces.ISession method), 195

`before_create()` (ckan.plugins.interfaces.IResourceController method), 199

`before_delete()` (ckan.plugins.interfaces.IMapper method), 194

`before_delete()` (ckan.plugins.interfaces.IResourceController method), 200

`before_flush()` (ckan.plugins.interfaces.ISession method), 194

`before_index()` (ckan.plugins.interfaces.IPackageController method), 199

`before_insert()` (ckan.plugins.interfaces.IMapper method), 194

`before_load()` (ckan.plugins.interfaces.IPluginObserver method), 200

`before_map()` (ckan.plugins.interfaces.IRoutes method), 194

`before_search()` (ckan.plugins.interfaces.IPackageController
method), 198
`before_show()` (ckan.plugins.interfaces.IResourceController
method), 200
`before_unload()` (ckan.plugins.interfaces.IPluginObserver
method), 200
`before_update()` (ckan.plugins.interfaces.IMapper
method), 194
`before_update()` (ckan.plugins.interfaces.IResourceController
method), 199
`before_view()` (ckan.plugins.interfaces.IGroupController
method), 198
`before_view()` (ckan.plugins.interfaces.IOrganizationController
method), 198
`before_view()` (ckan.plugins.interfaces.IPackageController
method), 199
`before_view()` (ckan.plugins.interfaces.ITagController
method), 198
`boolean_validator()` (in module ckan.logic.validators),
215
`build_extra_admin_nav()` (in module ckan.lib.helpers),
267
`build_nav()` (in module ckan.lib.helpers), 267
`build_nav_icon()` (in module ckan.lib.helpers), 267
`build_nav_main()` (in module ckan.lib.helpers), 267
`bulk_update_delete()` (in module
ckan.logic.action.update), 160
`bulk_update_private()` (in module
ckan.logic.action.update), 160
`bulk_update_public()` (in module
ckan.logic.action.update), 160
`button_attr()` (in module ckan.lib.helpers), 269

C

`c` (built-in variable), 263
`c` (ckan.plugins.toolkit attribute), 211
`call_action()` (in module ckan.tests.helpers), 326
`call_auth()` (in module ckan.tests.helpers), 326
`can_preview()` (ckan.plugins.interfaces.IResourcePreview
method), 197
`can_view()` (ckan.plugins.interfaces.IResourceView
method), 196
`change_config()` (in module ckan.tests.helpers), 327
`check_access()` (in module ckan.lib.helpers), 268
`check_config_permission()` (in module ckan.lib.helpers),
273
`check_data_dict()` (ckan.plugins.interfaces.IGroupForm
method), 207
`ckan.lib.helpers` (module), 265
`ckan.logic.action.create` (module), 148
`ckan.logic.action.delete` (module), 162
`ckan.logic.action.get` (module), 131
`ckan.logic.action.patch` (module), 161
`ckan.logic.action.update` (module), 156
`ckan.logic.converters` (module), 218
`ckan.logic.validators` (module), 215
`ckan.plugins.interfaces` (module), 193
`ckan.plugins.toolkit._()` (built-in function), 210
`ckan.plugins.toolkit.abort()` (built-in function), 210
`ckan.plugins.toolkit.add_ckan_admin_tab()` (built-in
function), 210
`ckan.plugins.toolkit.add_public_directory()` (built-in
function), 210
`ckan.plugins.toolkit.add_resource()` (built-in function),
210
`ckan.plugins.toolkit.add_template_directory()` (built-in
function), 210
`ckan.plugins.toolkit.asbool()` (built-in function), 211
`ckan.plugins.toolkit.asint()` (built-in function), 211
`ckan.plugins.toolkit.aslist()` (built-in function), 211
`ckan.plugins.toolkit.auth_allow_anonymous_access()`
(built-in function), 211
`ckan.plugins.toolkit.auth_disallow_anonymous_access()`
(built-in function), 211
`ckan.plugins.toolkit.auth_sysadmins_check()` (built-in
function), 211
`ckan.plugins.toolkit.BaseController` (built-in class), 209
`ckan.plugins.toolkit.check_access()` (built-in function),
211
`ckan.plugins.toolkit.check_ckan_version()` (built-in func-
tion), 212
`ckan.plugins.toolkit.CkanCommand` (built-in class), 209
`ckan.plugins.toolkit.CkanVersionException` (built-in
class), 209
`ckan.plugins.toolkit.DefaultDatasetForm` (built-in class),
209
`ckan.plugins.toolkit.DefaultGroupForm` (built-in class),
209
`ckan.plugins.toolkit.get_action()` (built-in function), 212
`ckan.plugins.toolkit.get_converter()` (built-in function),
212
`ckan.plugins.toolkit.get_or_bust()` (built-in function), 213
`ckan.plugins.toolkit.get_validator()` (built-in function),
213
`ckan.plugins.toolkit.Invalid` (built-in class), 210
`ckan.plugins.toolkit.literal` (built-in class), 213
`ckan.plugins.toolkit.navl_validate()` (built-in function),
213
`ckan.plugins.toolkit.NotAuthorized` (built-in class), 210
`ckan.plugins.toolkit.ObjectNotFound` (built-in class), 210
`ckan.plugins.toolkit.redirect_to()` (built-in function), 213
`ckan.plugins.toolkit.render()` (built-in function), 214
`ckan.plugins.toolkit.render_snippet()` (built-in function),
214
`ckan.plugins.toolkit.render_text()` (built-in function), 214
`ckan.plugins.toolkit.requires_ckan_version()` (built-in
function), 214
`ckan.plugins.toolkit.side_effect_free()` (built-in function),

- 214
 ckan.plugins.toolkit.UnknownValidator (built-in class), 210
 ckan.plugins.toolkit.url_for() (built-in function), 215
 ckan.plugins.toolkit.ValidationError (built-in class), 210
 ckan.tests.controllers (module), 332
 ckan.tests.factories (module), 324
 ckan.tests.helpers (module), 325
 ckan.tests.lib (module), 333
 ckan.tests.logic.action (module), 329
 ckan.tests.logic.auth (module), 330
 ckan.tests.logic.test_schema (module), 332
 ckan.tests.migration (module), 334
 ckan.tests.model (module), 333
 ckan.tests.plugins (module), 333
 ckanext.datastore.logic.action (module), 71
 clean_format() (in module ckan.logic.validators), 218
 config_option_list() (in module ckan.logic.action.get), 148
 config_option_show() (in module ckan.logic.action.get), 147
 config_option_update() (in module ckan.logic.action.update), 160
 configure() (ckan.plugins.interfaces.IConfigurable method), 200
 convert_from_extras() (in module ckan.logic.converters), 218
 convert_from_tags() (in module ckan.logic.converters), 218
 convert_group_name_or_id_to_id() (in module ckan.logic.converters), 218
 convert_package_name_or_id_to_id() (in module ckan.logic.converters), 218
 convert_to_dict() (in module ckan.lib.helpers), 270
 convert_to_extras() (in module ckan.logic.converters), 218
 convert_to_json_if_string() (in module ckan.logic.converters), 219
 convert_to_list_if_string() (in module ckan.logic.converters), 219
 convert_to_tags() (in module ckan.logic.converters), 218
 convert_user_name_or_id_to_id() (in module ckan.logic.converters), 218
 create_package_schema() (ckan.plugins.interfaces.IDatasetForm method), 203
 current_package_list_with_resources() (in module ckan.logic.action.get), 131
 current_url() (in module ckan.lib.helpers), 266
- D**
 dashboard_activity_list() (in module ckan.logic.action.get), 146
 dashboard_activity_list_html() (in module ckan.logic.action.get), 147
 dashboard_activity_stream() (in module ckan.lib.helpers), 271
 dashboard_mark_activities_old() (in module ckan.logic.action.update), 160
 dashboard_new_activities_count() (in module ckan.logic.action.get), 147
 Dataset (class in ckan.tests.factories), 325
 dataset_display_name() (in module ckan.lib.helpers), 269
 dataset_facets() (ckan.plugins.interfaces.IFacets method), 208
 dataset_follower_count() (in module ckan.logic.action.get), 145
 dataset_follower_list() (in module ckan.logic.action.get), 146
 dataset_follower_count() (in module ckan.logic.action.get), 144
 dataset_follower_list() (in module ckan.logic.action.get), 145
 dataset_link() (in module ckan.lib.helpers), 269
 datasets_with_no_organization_cannot_be_private() (in module ckan.logic.validators), 217
 datastore_create() (in module ckanext.datastore.logic.action), 71
 datastore_delete() (in module ckanext.datastore.logic.action), 72
 datastore_info() (in module ckanext.datastore.logic.action), 72
 datastore_make_private() (in module ckanext.datastore.logic.action), 74
 datastore_make_public() (in module ckanext.datastore.logic.action), 74
 datastore_search() (in module ckanext.datastore.logic.action), 73
 datastore_search_sql() (in module ckanext.datastore.logic.action), 73
 datastore_upsert() (in module ckanext.datastore.logic.action), 72
 date_str_to_datetime() (in module ckan.lib.helpers), 269
 db_to_form_schema() (ckan.plugins.interfaces.IGroupForm method), 207
 debug_full_info_as_list() (in module ckan.lib.helpers), 271
 debug_inspect() (in module ckan.lib.helpers), 270
 default_group_type() (in module ckan.lib.helpers), 267
 dict_list_reduce() (in module ckan.lib.helpers), 268
 dump_json() (in module ckan.lib.helpers), 269
 duplicate_extras_key() (in module ckan.logic.validators), 216
- E**
 edit_template() (ckan.plugins.interfaces.IDatasetForm method), 204

- [edit_template\(\)](#) (ckan.plugins.interfaces.IGroupForm method), 206
[empty_if_not_sysadmin\(\)](#) (in module ckan.logic.validators), 218
[escape_js\(\)](#) (in module ckan.lib.helpers), 271
[extra_key_not_in_root_schema\(\)](#) (in module ckan.logic.validators), 218
[extras_unicode_convert\(\)](#) (in module ckan.logic.converters), 218
- ## F
- [FACTORY_FOR](#) (ckan.tests.factories.MockUser attribute), 325
[featured_group_org\(\)](#) (in module ckan.lib.helpers), 273
[filter\(\)](#) (ckan.plugins.interfaces.IGenshiStreamFilter method), 193
[filter_fields_and_values_exist_and_are_valid\(\)](#) (in module ckan.logic.validators), 218
[filter_fields_and_values_should_have_same_length\(\)](#) (in module ckan.logic.validators), 218
[flash_error\(\)](#) (in module ckan.lib.helpers), 266
[flash_notice\(\)](#) (in module ckan.lib.helpers), 266
[flash_success\(\)](#) (in module ckan.lib.helpers), 266
[follow_button\(\)](#) (in module ckan.lib.helpers), 270
[follow_count\(\)](#) (in module ckan.lib.helpers), 270
[follow_dataset\(\)](#) (in module ckan.logic.action.create), 155
[follow_group\(\)](#) (in module ckan.logic.action.create), 156
[follow_user\(\)](#) (in module ckan.logic.action.create), 155
[followee_count\(\)](#) (in module ckan.logic.action.get), 145
[followee_list\(\)](#) (in module ckan.logic.action.get), 146
[form_template\(\)](#) (ckan.plugins.interfaces.IResourceView method), 197
[form_to_db_schema\(\)](#) (ckan.plugins.interfaces.IGroupForm method), 207
[format_autocomplete\(\)](#) (in module ckan.logic.action.get), 137
[format_icon\(\)](#) (in module ckan.lib.helpers), 268
[format_resource_items\(\)](#) (in module ckan.lib.helpers), 272
[free_tags_only\(\)](#) (in module ckan.logic.converters), 218
[full_current_url\(\)](#) (in module ckan.lib.helpers), 266
[FunctionalTestBase](#) (class in ckan.tests.helpers), 326
- ## G
- [g](#) (built-in variable), 263
[get_actions\(\)](#) (ckan.plugins.interfaces.IActions method), 201
[get_allowed_view_types\(\)](#) (in module ckan.lib.helpers), 272
[get_auth_functions\(\)](#) (ckan.plugins.interfaces.IAuthFunctions method), 201
[get_facet_items_dict\(\)](#) (in module ckan.lib.helpers), 267
[get_featured_groups\(\)](#) (in module ckan.lib.helpers), 273
[get_featured_organizations\(\)](#) (in module ckan.lib.helpers), 273
[get_helpers\(\)](#) (ckan.plugins.interfaces.ITemplateHelpers method), 202
[get_organization\(\)](#) (in module ckan.lib.helpers), 273
[get_param_int\(\)](#) (in module ckan.lib.helpers), 268
[get_pkg_dict_extra\(\)](#) (in module ckan.lib.helpers), 271
[get_request_param\(\)](#) (in module ckan.lib.helpers), 271
[get_site_protocol_and_host\(\)](#) (in module ckan.lib.helpers), 265
[get_site_statistics\(\)](#) (in module ckan.lib.helpers), 273
[get_site_user\(\)](#) (in module ckan.logic.action.get), 141
[get_validators\(\)](#) (ckan.plugins.interfaces.IValidators method), 201
[gravatar\(\)](#) (in module ckan.lib.helpers), 268
[Group](#) (class in ckan.tests.factories), 325
[group_activity_list\(\)](#) (in module ckan.logic.action.get), 142
[group_activity_list_html\(\)](#) (in module ckan.logic.action.get), 143
[group_create\(\)](#) (in module ckan.logic.action.create), 152
[group_delete\(\)](#) (in module ckan.logic.action.delete), 163
[group_facets\(\)](#) (ckan.plugins.interfaces.IFacets method), 208
[group_follower_count\(\)](#) (in module ckan.logic.action.get), 146
[group_follower_list\(\)](#) (in module ckan.logic.action.get), 146
[group_follower_count\(\)](#) (in module ckan.logic.action.get), 144
[group_follower_list\(\)](#) (in module ckan.logic.action.get), 145
[group_id_exists\(\)](#) (in module ckan.logic.validators), 216
[group_id_or_name_exists\(\)](#) (in module ckan.logic.validators), 216
[group_link\(\)](#) (in module ckan.lib.helpers), 269
[group_list\(\)](#) (in module ckan.logic.action.get), 132
[group_list_authz\(\)](#) (in module ckan.logic.action.get), 133
[group_member_create\(\)](#) (in module ckan.logic.action.create), 155
[group_member_delete\(\)](#) (in module ckan.logic.action.delete), 164
[group_name_to_title\(\)](#) (in module ckan.lib.helpers), 268
[group_name_validator\(\)](#) (in module ckan.logic.validators), 216
[group_package_show\(\)](#) (in module ckan.logic.action.get), 136
[group_patch\(\)](#) (in module ckan.logic.action.patch), 162
[group_purge\(\)](#) (in module ckan.logic.action.delete), 163
[group_revision_list\(\)](#) (in module ckan.logic.action.get), 134
[group_show\(\)](#) (in module ckan.logic.action.get), 135
[group_types\(\)](#) (ckan.plugins.interfaces.IGroupForm method), 206

group_update() (in module ckan.logic.action.update), 157
groups_available() (in module ckan.lib.helpers), 271

H

h (built-in variable), 263
has_more_facets() (in module ckan.lib.helpers), 267
help_show() (in module ckan.logic.action.get), 147
history_template() (ckan.plugins.interfaces.IDatasetForm method), 205
history_template() (ckan.plugins.interfaces.IGroupForm method), 206
html_auto_link() (in module ckan.lib.helpers), 271

I

IActions (class in ckan.plugins.interfaces), 201
IA Authenticator (class in ckan.plugins.interfaces), 209
IAuthFunctions (class in ckan.plugins.interfaces), 201
icon() (in module ckan.lib.helpers), 268
icon_html() (in module ckan.lib.helpers), 268
icon_url() (in module ckan.lib.helpers), 268
IConfigurable (class in ckan.plugins.interfaces), 200
IConfigurer (class in ckan.plugins.interfaces), 200
IDatasetForm (class in ckan.plugins.interfaces), 202
identify() (ckan.plugins.interfaces.IAuthenticator method), 209
IDomainObjectModification (class in ckan.plugins.interfaces), 195
if_empty_guess_format() (in module ckan.logic.validators), 218
IFacets (class in ckan.plugins.interfaces), 207
IGenshiStreamFilter (class in ckan.plugins.interfaces), 193
ignore_not_admin() (in module ckan.logic.validators), 217
ignore_not_group_admin() (in module ckan.logic.validators), 217
ignore_not_package_admin() (in module ckan.logic.validators), 217
ignore_not_sysadmin() (in module ckan.logic.validators), 217
IGroupController (class in ckan.plugins.interfaces), 198
IGroupForm (class in ckan.plugins.interfaces), 206
IMapper (class in ckan.plugins.interfaces), 194
IMiddleware (class in ckan.plugins.interfaces), 193
implements() (in module ckan.plugins), 193
include_resource() (in module ckan.lib.helpers), 270
index_template() (ckan.plugins.interfaces.IGroupForm method), 206
info() (ckan.plugins.interfaces.IResourceView method), 195
int_validator() (in module ckan.logic.validators), 215
IOrganizationController (class in ckan.plugins.interfaces), 198
IPackageController (class in ckan.plugins.interfaces), 198

IPluginObserver (class in ckan.plugins.interfaces), 200
IResourceController (class in ckan.plugins.interfaces), 199
IResourcePreview (class in ckan.plugins.interfaces), 197
IResourceUrlChange (class in ckan.plugins.interfaces), 195
IResourceView (class in ckan.plugins.interfaces), 195
IRoutes (class in ckan.plugins.interfaces), 194
is_fallback() (ckan.plugins.interfaces.IDatasetForm method), 203
is_fallback() (ckan.plugins.interfaces.IGroupForm method), 206
is_positive_integer() (in module ckan.logic.validators), 215
is_url() (in module ckan.lib.helpers), 266
ISession (class in ckan.plugins.interfaces), 194
isodate() (in module ckan.logic.validators), 215
ITagController (class in ckan.plugins.interfaces), 198
ITemplateHelpers (class in ckan.plugins.interfaces), 202
IValidators (class in ckan.plugins.interfaces), 201

L

lang() (in module ckan.lib.helpers), 266
lang_native_name() (in module ckan.lib.helpers), 266
license_list() (in module ckan.logic.action.get), 134
license_options() (in module ckan.lib.helpers), 273
linked_gravatar() (in module ckan.lib.helpers), 268
linked_user() (in module ckan.lib.helpers), 268
list_dict_filter() (in module ckan.lib.helpers), 272
list_of_strings() (in module ckan.logic.validators), 217
login() (ckan.plugins.interfaces.IAuthenticator method), 209
logout() (ckan.plugins.interfaces.IAuthenticator method), 209

M

mail_to() (in module ckan.lib.helpers), 273
make_error_log_middleware() (ckan.plugins.interfaces.IMiddleware method), 193
make_middleware() (ckan.plugins.interfaces.IMiddleware method), 193
markdown_extract() (in module ckan.lib.helpers), 268
member_create() (in module ckan.logic.action.create), 151
member_delete() (in module ckan.logic.action.delete), 163
member_list() (in module ckan.logic.action.get), 132
member_roles_list() (in module ckan.logic.action.get), 147
Message (class in ckan.lib.helpers), 266
missing (ckan.plugins.toolkit attribute), 213
MockUser (class in ckan.tests.factories), 325

N

`N_()` (built-in function), 263
`name_validator()` (in module `ckan.logic.validators`), 216
`natural_number_validator()` (in module `ckan.logic.validators`), 215
`nav_link()` (in module `ckan.lib.helpers`), 266
`new_activities()` (in module `ckan.lib.helpers`), 272
`new_template()` (`ckan.plugins.interfaces.IDatasetForm` method), 204
`new_template()` (`ckan.plugins.interfaces.IGroupForm` method), 206
`no_http()` (in module `ckan.logic.validators`), 215
`no_loops_in_hierarchy()` (in module `ckan.logic.validators`), 218

O

`object_id_validator()` (in module `ckan.logic.validators`), 216
`Organization` (class in `ckan.tests.factories`), 325
`organization_activity_list()` (in module `ckan.logic.action.get`), 143
`organization_activity_list_html()` (in module `ckan.logic.action.get`), 144
`organization_autocomplete()` (in module `ckan.logic.action.get`), 137
`organization_create()` (in module `ckan.logic.action.create`), 152
`organization_delete()` (in module `ckan.logic.action.delete`), 163
`organization_facets()` (`ckan.plugins.interfaces.IFacets` method), 208
`organization_follower_list()` (in module `ckan.logic.action.get`), 146
`organization_follower_count()` (in module `ckan.logic.action.get`), 144
`organization_follower_list()` (in module `ckan.logic.action.get`), 145
`organization_link()` (in module `ckan.lib.helpers`), 269
`organization_list()` (in module `ckan.logic.action.get`), 132
`organization_list_for_user()` (in module `ckan.logic.action.get`), 133
`organization_member_create()` (in module `ckan.logic.action.create`), 155
`organization_member_delete()` (in module `ckan.logic.action.delete`), 164
`organization_patch()` (in module `ckan.logic.action.patch`), 162
`organization_purge()` (in module `ckan.logic.action.delete`), 163
`organization_revision_list()` (in module `ckan.logic.action.get`), 134
`organization_show()` (in module `ckan.logic.action.get`), 136

`organization_update()` (in module `ckan.logic.action.update`), 158
`organizations_available()` (in module `ckan.lib.helpers`), 271
`owner_org_validator()` (in module `ckan.logic.validators`), 215

P

`package_activity_list()` (in module `ckan.logic.action.get`), 142
`package_activity_list_html()` (in module `ckan.logic.action.get`), 143
`package_autocomplete()` (in module `ckan.logic.action.get`), 137
`package_create()` (in module `ckan.logic.action.create`), 148
`package_create_default_resource_views()` (in module `ckan.logic.action.create`), 150
`package_delete()` (in module `ckan.logic.action.delete`), 162
`package_form()` (`ckan.plugins.interfaces.IDatasetForm` method), 205
`package_form()` (`ckan.plugins.interfaces.IGroupForm` method), 207
`package_id_does_not_exist()` (in module `ckan.logic.validators`), 215
`package_id_exists()` (in module `ckan.logic.validators`), 215
`package_id_not_changed()` (in module `ckan.logic.validators`), 215
`package_id_or_name_exists()` (in module `ckan.logic.validators`), 215
`package_list()` (in module `ckan.logic.action.get`), 131
`package_name_exists()` (in module `ckan.logic.validators`), 215
`package_name_validator()` (in module `ckan.logic.validators`), 216
`package_owner_org_update()` (in module `ckan.logic.action.update`), 160
`package_patch()` (in module `ckan.logic.action.patch`), 161
`package_relationship_create()` (in module `ckan.logic.action.create`), 151
`package_relationship_delete()` (in module `ckan.logic.action.delete`), 162
`package_relationship_update()` (in module `ckan.logic.action.update`), 157
`package_relationships_list()` (in module `ckan.logic.action.get`), 134
`package_resource_reorder()` (in module `ckan.logic.action.update`), 157
`package_revision_list()` (in module `ckan.logic.action.get`), 131
`package_search()` (in module `ckan.logic.action.get`), 138
`package_show()` (in module `ckan.logic.action.get`), 135

- [package_types\(\)](#) (ckan.plugins.interfaces.IDatasetForm method), 203
[package_update\(\)](#) (in module ckan.logic.action.update), 157
[package_version_validator\(\)](#) (in module ckan.logic.validators), 216
[Page](#) (class in ckan.lib.helpers), 268
[pager\(\)](#) (ckan.lib.helpers.Page method), 268
[pager_url\(\)](#) (in module ckan.lib.helpers), 268
[parse_rfc_2822_date\(\)](#) (in module ckan.lib.helpers), 269
[Plugin](#) (class in ckan.plugins), 193
[popular\(\)](#) (in module ckan.lib.helpers), 271
[preview_template\(\)](#) (ckan.plugins.interfaces.IResourcePreview method), 197
- ## R
- [rating_create\(\)](#) (in module ckan.logic.action.create), 153
[read_template\(\)](#) (ckan.plugins.interfaces.IDatasetForm method), 204
[read_template\(\)](#) (ckan.plugins.interfaces.IGroupForm method), 206
[recently_changed_packages_activity_list\(\)](#) (in module ckan.logic.action.get), 143
[recently_changed_packages_activity_list_html\(\)](#) (in module ckan.logic.action.get), 144
[recently_changed_packages_activity_stream\(\)](#) (in module ckan.lib.helpers), 271
[redirect_to\(\)](#) (in module ckan.lib.helpers), 265
[Related](#) (class in ckan.tests.factories), 325
[related_create\(\)](#) (in module ckan.logic.action.create), 151
[related_delete\(\)](#) (in module ckan.logic.action.delete), 163
[related_id_exists\(\)](#) (in module ckan.logic.validators), 216
[related_item_link\(\)](#) (in module ckan.lib.helpers), 269
[related_list\(\)](#) (in module ckan.logic.action.get), 131
[related_show\(\)](#) (in module ckan.logic.action.get), 131
[related_update\(\)](#) (in module ckan.logic.action.update), 156
[remove_linebreaks\(\)](#) (in module ckan.lib.helpers), 272
[remove_url_param\(\)](#) (in module ckan.lib.helpers), 270
[remove_whitespace\(\)](#) (in module ckan.logic.converters), 219
[render_datetime\(\)](#) (in module ckan.lib.helpers), 268
[render_markdown\(\)](#) (in module ckan.lib.helpers), 272
[rendered_resource_view\(\)](#) (in module ckan.lib.helpers), 272
[request](#) (built-in variable), 263
[request](#) (ckan.plugins.toolkit attribute), 214
[reset_db\(\)](#) (in module ckan.tests.helpers), 325
[Resource](#) (class in ckan.tests.factories), 324
[resource_create\(\)](#) (in module ckan.logic.action.create), 149
[resource_create_default_resource_views\(\)](#) (in module ckan.logic.action.create), 150
[resource_delete\(\)](#) (in module ckan.logic.action.delete), 162
[resource_display_name\(\)](#) (in module ckan.lib.helpers), 269
[resource_form\(\)](#) (ckan.plugins.interfaces.IDatasetForm method), 205
[resource_formats\(\)](#) (in module ckan.lib.helpers), 273
[resource_icon\(\)](#) (in module ckan.lib.helpers), 268
[resource_id_exists\(\)](#) (in module ckan.logic.validators), 216
[resource_link\(\)](#) (in module ckan.lib.helpers), 269
[resource_patch\(\)](#) (in module ckan.logic.action.patch), 161
[resource_preview\(\)](#) (in module ckan.lib.helpers), 272
[resource_search\(\)](#) (in module ckan.logic.action.get), 139
[resource_show\(\)](#) (in module ckan.logic.action.get), 135
[resource_status_show\(\)](#) (in module ckan.logic.action.get), 135
[resource_template\(\)](#) (ckan.plugins.interfaces.IDatasetForm method), 205
[resource_update\(\)](#) (in module ckan.logic.action.update), 156
[resource_view_clear\(\)](#) (in module ckan.logic.action.delete), 162
[resource_view_create\(\)](#) (in module ckan.logic.action.create), 150
[resource_view_delete\(\)](#) (in module ckan.logic.action.delete), 162
[resource_view_display_preview\(\)](#) (in module ckan.lib.helpers), 272
[resource_view_full_page\(\)](#) (in module ckan.lib.helpers), 272
[resource_view_get_fields\(\)](#) (in module ckan.lib.helpers), 272
[resource_view_icon\(\)](#) (in module ckan.lib.helpers), 272
[resource_view_is_filterable\(\)](#) (in module ckan.lib.helpers), 272
[resource_view_is_iframed\(\)](#) (in module ckan.lib.helpers), 272
[resource_view_list\(\)](#) (in module ckan.logic.action.get), 135
[resource_view_reorder\(\)](#) (in module ckan.logic.action.update), 156
[resource_view_show\(\)](#) (in module ckan.logic.action.get), 135
[resource_view_update\(\)](#) (in module ckan.logic.action.update), 156
[ResourceView](#) (class in ckan.tests.factories), 324
[response](#) (built-in variable), 263
[response](#) (ckan.plugins.toolkit attribute), 214
[revision_list\(\)](#) (in module ckan.logic.action.get), 131
[revision_show\(\)](#) (in module ckan.logic.action.get), 135
[role_exists\(\)](#) (in module ckan.logic.validators), 217
[roles_show\(\)](#) (in module ckan.logic.action.get), 141

S

`sanitize_id()` (in module `ckan.lib.helpers`), 273
`search_template()` (`ckan.plugins.interfaces.IDatasetForm` method), 205
`send_email_notifications()` (in module `ckan.logic.action.update`), 160
`session` (built-in variable), 263
`setup()` (`ckan.tests.helpers.FunctionalTestBase` method), 327
`setup_template_variables()` (`ckan.plugins.interfaces.IDatasetForm` method), 204
`setup_template_variables()` (`ckan.plugins.interfaces.IGroupForm` method), 207
`setup_template_variables()` (`ckan.plugins.interfaces.IResourcePreview` method), 197
`setup_template_variables()` (`ckan.plugins.interfaces.IResourceView` method), 196
`show_package_schema()` (`ckan.plugins.interfaces.IDatasetForm` method), 204
`SI_number_span()` (in module `ckan.lib.helpers`), 272
`SingletonPlugin` (class in `ckan.plugins`), 193
`site_read()` (in module `ckan.logic.action.get`), 131
`snippet()` (in module `ckan.lib.helpers`), 269
`sorted_extras()` (in module `ckan.lib.helpers`), 268
`status_show()` (in module `ckan.logic.action.get`), 142
`submit_and_follow()` (in module `ckan.tests.helpers`), 327
`Sysadmin` (class in `ckan.tests.factories`), 325
`SystemInfo` (class in `ckan.tests.factories`), 325

T

`tag_autocomplete()` (in module `ckan.logic.action.get`), 141
`tag_create()` (in module `ckan.logic.action.create`), 154
`tag_delete()` (in module `ckan.logic.action.delete`), 164
`tag_in_vocabulary_validator()` (in module `ckan.logic.validators`), 217
`tag_length_validator()` (in module `ckan.logic.validators`), 216
`tag_link()` (in module `ckan.lib.helpers`), 269
`tag_list()` (in module `ckan.logic.action.get`), 134
`tag_name_validator()` (in module `ckan.logic.validators`), 217
`tag_not_in_vocabulary()` (in module `ckan.logic.validators`), 217
`tag_not_uppercase()` (in module `ckan.logic.validators`), 217
`tag_search()` (in module `ckan.logic.action.get`), 140
`tag_show()` (in module `ckan.logic.action.get`), 136

`tag_string_convert()` (in module `ckan.logic.validators`), 217
`task_status_delete()` (in module `ckan.logic.action.delete`), 163
`task_status_show()` (in module `ckan.logic.action.get`), 141
`task_status_update()` (in module `ckan.logic.action.update`), 158
`task_status_update_many()` (in module `ckan.logic.action.update`), 159
`term_translation_show()` (in module `ckan.logic.action.get`), 141
`term_translation_update()` (in module `ckan.logic.action.update`), 159
`term_translation_update_many()` (in module `ckan.logic.action.update`), 159
`time_ago_from_timestamp()` (in module `ckan.lib.helpers`), 269
`tmpl_context` (built-in variable), 262
`translator` (built-in variable), 264

U

`unfollow_dataset()` (in module `ckan.logic.action.delete`), 164
`unfollow_group()` (in module `ckan.logic.action.delete`), 164
`unfollow_user()` (in module `ckan.logic.action.delete`), 164
`ungettext()` (built-in function), 263
`unified_resource_format()` (in module `ckan.lib.helpers`), 273
`unselected_facet_items()` (in module `ckan.lib.helpers`), 268
`update_config()` (`ckan.plugins.interfaces.IConfigurer` method), 200
`update_config_schema()` (`ckan.plugins.interfaces.IConfigurer` method), 200
`update_package_schema()` (`ckan.plugins.interfaces.IDatasetForm` method), 203
`uploads_enabled()` (in module `ckan.lib.helpers`), 273
`url()` (in module `ckan.lib.helpers`), 265
`url_for()` (in module `ckan.lib.helpers`), 265
`url_for_static()` (in module `ckan.lib.helpers`), 266
`url_for_static_or_external()` (in module `ckan.lib.helpers`), 266
`url_is_local()` (in module `ckan.lib.helpers`), 266
`url_validator()` (in module `ckan.logic.validators`), 217
`urls_for_resource()` (in module `ckan.lib.helpers`), 270
`User` (class in `ckan.tests.factories`), 324
`user_about_validator()` (in module `ckan.logic.validators`), 217
`user_activity_list()` (in module `ckan.logic.action.get`), 142
`user_activity_list_html()` (in module `ckan.logic.action.get`), 143

[user_autocomplete\(\)](#) (in module `ckan.logic.action.get`), [137](#)
[user_both_passwords_entered\(\)](#) (in module `ckan.logic.validators`), [217](#)
[user_create\(\)](#) (in module `ckan.logic.action.create`), [153](#)
[user_delete\(\)](#) (in module `ckan.logic.action.delete`), [162](#)
[user_followee_count\(\)](#) (in module `ckan.logic.action.get`), [145](#)
[user_followee_list\(\)](#) (in module `ckan.logic.action.get`), [146](#)
[user_follower_count\(\)](#) (in module `ckan.logic.action.get`), [144](#)
[user_follower_list\(\)](#) (in module `ckan.logic.action.get`), [145](#)
[user_generate_apikey\(\)](#) (in module `ckan.logic.action.update`), [158](#)
[user_id_exists\(\)](#) (in module `ckan.logic.validators`), [216](#)
[user_id_or_name_exists\(\)](#) (in module `ckan.logic.validators`), [216](#)
[user_in_org_or_group\(\)](#) (in module `ckan.lib.helpers`), [271](#)
[user_invite\(\)](#) (in module `ckan.logic.action.create`), [154](#)
[user_list\(\)](#) (in module `ckan.logic.action.get`), [134](#)
[user_name_exists\(\)](#) (in module `ckan.logic.validators`), [217](#)
[user_name_validator\(\)](#) (in module `ckan.logic.validators`), [217](#)
[user_password_not_empty\(\)](#) (in module `ckan.logic.validators`), [217](#)
[user_password_validator\(\)](#) (in module `ckan.logic.validators`), [217](#)
[user_passwords_match\(\)](#) (in module `ckan.logic.validators`), [217](#)
[user_role_bulk_update\(\)](#) (in module `ckan.logic.action.update`), [160](#)
[user_role_update\(\)](#) (in module `ckan.logic.action.update`), [159](#)
[user_show\(\)](#) (in module `ckan.logic.action.get`), [137](#)
[user_update\(\)](#) (in module `ckan.logic.action.update`), [158](#)
[vocabulary_delete\(\)](#) (in module `ckan.logic.action.delete`), [164](#)
[vocabulary_id_exists\(\)](#) (in module `ckan.logic.validators`), [217](#)
[vocabulary_id_not_changed\(\)](#) (in module `ckan.logic.validators`), [217](#)
[vocabulary_list\(\)](#) (in module `ckan.logic.action.get`), [142](#)
[vocabulary_name_validator\(\)](#) (in module `ckan.logic.validators`), [217](#)
[vocabulary_show\(\)](#) (in module `ckan.logic.action.get`), [142](#)
[vocabulary_update\(\)](#) (in module `ckan.logic.action.update`), [159](#)

W

[webtest_submit\(\)](#) (in module `ckan.tests.helpers`), [327](#)
[webtest_submit_fields\(\)](#) (in module `ckan.tests.helpers`), [327](#)

V

[validate\(\)](#) (`ckan.plugins.interfaces.IDatasetForm` method), [205](#)
[validate\(\)](#) (`ckan.plugins.interfaces.IGroupForm` method), [207](#)
[validator_data_dict\(\)](#) (in module `ckan.tests.factories`), [325](#)
[validator_errors_dict\(\)](#) (in module `ckan.tests.factories`), [325](#)
[view_resource_url\(\)](#) (in module `ckan.lib.helpers`), [272](#)
[view_template\(\)](#) (`ckan.plugins.interfaces.IResourceView` method), [196](#)
[Vocabulary](#) (class in `ckan.tests.factories`), [325](#)
[vocabulary_create\(\)](#) (in module `ckan.logic.action.create`), [154](#)