Docker and Docker Compose installation

Overview

Contains Docker images for the different components of CKAN Cloud and a Docker compose environment (based on ckan) for development and testing Open Data portals.

TIP

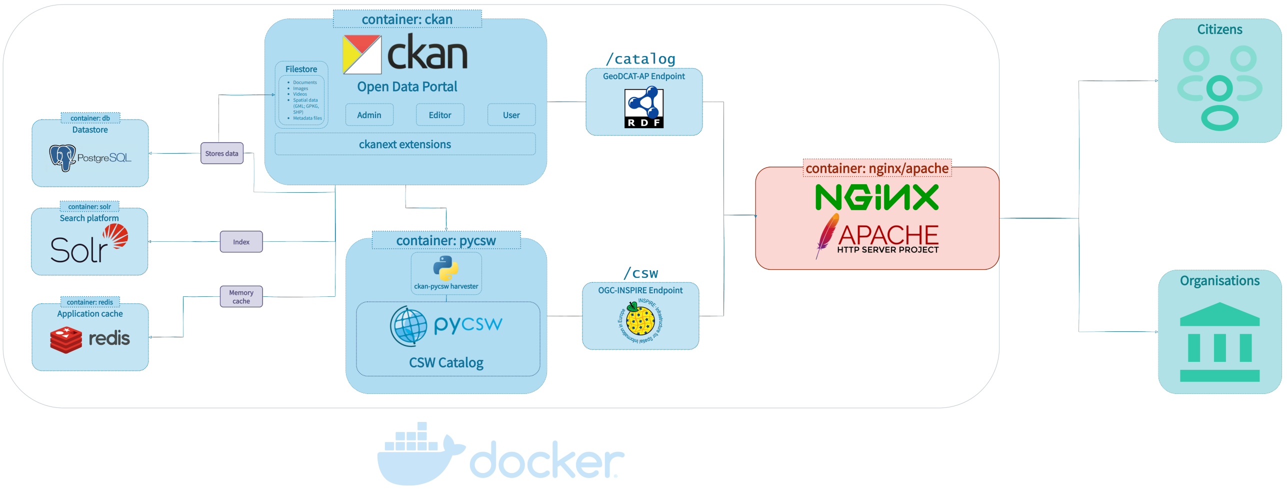
- Use the **deploy in 5 minutes** to see `ckan-docker` in **5 minutes** ✨!
- Or use Codespaces to test `ckan-docker` in your browser:

Open in GitHub Codespaces

WARNING

This is a **custom installation of Docker Compose** with specific extensions for spatial data and GeoDCAT-AP/INSPIRE metadata profiles. For official installations, please have a look: CKAN: Source installation.

Available components



Available components:

- CKAN custom multi-stage build with spatial capabilities from ckan-docker-spatial, an image used as a base and built from the official CKAN repo. The following versions of CKAN are available:

CKAN Version	Type	Docker tag	Notes
2.9.8	custom image	`ghcr.io/mjanez/ckan-spatial:ckan-2.9.8`	Stable version with CKAN 2.9.8
2.9.9	custom image	`ghcr.io/mjanez/ckan-docker:ckan-2.9.9`	Stable version with CKAN 2.9.9
2.9.10	custom image	`ghcr.io/mjanez/ckan-docker:ckan-2.9.10`	Stable version with CKAN 2.9.10

CKAN Version	Type	Docker tag	Notes
2.9.11	custom image	`ghcr.io/mjanez/ckan-docker:ckan-2.9.11`	Stable version with CKAN 2.9.11
2.9.11	latest custom image	`ghcr.io/mjanez/ckan-docker:master`	Latest `ckan-docker` image.

The non-CKAN images are as follows:

- PostgreSQL: Custom image based on official PostgreSQL image. Database files are stored in a named volume.
- Solr: Custom image based on official CKAN pre-configured Solr image. The index data is stored in a named volume and has a custom spatial schema upgrades. [^2]
- Redis: Standard Redis image
- NGINX: Latest stable nginx image that includes SSL and Non-SSL endpoints.
- ckan-pycsw: Custom image based on pycsw CKAN harvester ISO19139 for INSPIRE Metadata CSW Endpoint.

Optional HTTP Endpoint (`docker-compose.apache.yml`):

- `docker-compose.apache.yml`:
  - Apache HTTP Server: Custom image based on official latest stable httpd image. Configured to serve multiple routes for the ckan-pycsw CSW endpoint (`{CKAN_SITE_URL}/csw`) and CKAN (`{CKAN_SITE_URL}/catalog`). Only HTTP.

The site is configured using environment variables that you can set in the `.env` file for an NGINX and ckan-pycsw deployment (default `.env.example`), or replace it with the `.env.apache.example` for a Apache HTTP Server deployment using the Docker Compose file: `docker-compose.apache.yml`.

Requirements

`ckan-docker` consists of a set of containers deploy with docker and `docker compose`.

Prerequisites

docker compose *vs* docker-compose

All Docker Compose commands in this document will use the V2 version of Compose ie: `docker compose`. The older version (V1) used the `docker-compose` command. Please see Docker Compose for more information.

Install docker-engine

Follow the installation instructions for your environment to install Docker Engine.

To verify a successful Docker installation, run `docker run hello-world` and `docker version`. These commands should output versions for client and server.

Docker/Docker Compose more info

Learn more about Docker/Docker Compose basic commands.

CKAN: Installation and configuration

Clone and configure `.env`

The site is configured using environment variables that you can set in the `.env` file.

1. Clone project

   ```
   cd /path/to/my/project
   git clone https://github.com/mjanez/ckan-docker.git & cd ckan-docker
   ```

2. Copy the `.env.example` template and modify the resulting `.env` to suit your needs.

   ```
   cp .env.example .env
   ```

   NGINX **Apache HTTP Server**

   Modify the `.env` variables as needed:

   ```
   # Host Ports
   CKAN_PORT_HOST=5000
   NGINX_PORT_HOST=81
   NGINX_SSLPORT_HOST=8443
   APACHE_PORT_HOST=81
   PYCSW_PORT_HOST=8000

   ...

   #NGINX/APACHE
   ## Check CKAN__ROOT_PATH and CKANEXT__DCAT__BASE_URI and CKANEXT__SCHEMINGDCAT_GEOMETADATA_BASE_URI. If
   you don't need to use domain locations, it is better to use the nginx configuration. Leave blank or use
   the root `/`.
   PROXY_SERVER_NAME=localhost
   PROXY_CKAN_LOCATION=/catalog
   PROXY_PYCSW_LOCATION=/csw

   ...

   # CKAN_SITE_URL = http:/ or https:/ + PROXY_SERVER_NAME. Optionally the APACHE_HOST_PORT if different
   from 80
   CKAN_SITE_URL=http://localhost:81
   CKAN__ROOT_PATH=/catalog/{{LANG}}
   CKAN_PORT=5000
   CKAN__FAVICON=/catalog/base/images/ckan.ico
   CKAN__SITE_LOGO=/images/default/ckan-logo.png
   ```

   Modify the `.env` variables as needed:

```
# Host Ports
CKAN_PORT_HOST=5000
NGINX_PORT_HOST=81
NGINX_SSLPORT_HOST=8443
APACHE_PORT_HOST=81
PYCSW_PORT_HOST=8000

...

#NGINX/APACHE
## Check CKAN__ROOT_PATH and CKANEXT__DCAT__BASE_URI and CKANEXT__SCHEMINGDCAT_GEOMETADATA_BASE_URI. If
you don't need to use domain locations, it is better to use the nginx configuration. Leave blank or use
the root `/`.
PROXY_SERVER_NAME=localhost
PROXY_CKAN_LOCATION=/catalog
PROXY_PYCSW_LOCATION=/csw

...

# CKAN_SITE_URL = http:/ or https:/ + PROXY_SERVER_NAME. Optionally the APACHE_HOST_PORT if different
from 80
CKAN_SITE_URL=http://localhost:81
CKAN__ROOT_PATH=/catalog/{{LANG}}
CKAN_PORT=5000
CKAN__FAVICON=/catalog/base/images/ckan.ico
CKAN__SITE_LOGO=/images/default/ckan-logo.png
```

WARNING

Using the default values on the `.env` file will get you a working CKAN instance. There is a sysadmin user created by default with the values defined in `CKAN_SYSADMIN_NAME` and `CKAN_SYSADMIN_PASSWORD` (`ckan_admin` and `test1234` by default). All ennvars with `API_TOKEN` are automatically regenerated when CKAN is loaded, no editing is required.

This should be obviously changed before running this setup as a public CKAN instance.

You are now ready to proceed with deployment.

Install (build and run) CKAN plus dependencies

Base mode

Use this if you are a maintainer and will not be making code changes to CKAN or to CKAN extensions.

1. Build the images:

```
docker compose build
```

2. Start the containers:

```
docker compose up
```

This will start up the containers in the current window. By default the containers will log direct to this window with each container using a different colour. You could also use the -d "detach mode" option ie: `docker compose up -d` if you wished to use the current window for something else.

TIP

- Or build & up the containers:

  ```
  docker compose up -d --build
  ```

- Or use the Apache HTTP Server version:

  ```
  docker compose -f docker-compose.apache.yml up -d --build
  ```

Learn more about configuring this `ckan-docker`

- Backup the CKAN Database
- Configuring a docker compose service to start on boot

Quick mode

If you just want to test the package and see the general functionality of the platform, you can use the `ckan-docker` image from the Github container registry:

```
# Edit the envvars in the .env as you like and start the containers.
docker compose -f docker-compose.ghcr.yml up -d --build
```

INFO

It will download the pre-built image and deploy all the containers. Remember to use your own domain by changing `localhost` in the `.env` file.

Development mode

Use this mode if you are making code changes to CKAN and either creating new extensions or making code changes to existing extensions. This mode also uses the `.env` file for config options.

To develop local extensions use the `docker compose.dev.yml` file:

To build the images:

```
docker compose -f docker-compose.dev.yml build
```

To start the containers:

```
docker compose -f docker-compose.dev.yml up
```

TIP

See CKAN images for more details of what happens when using development mode.

Check the containers are running

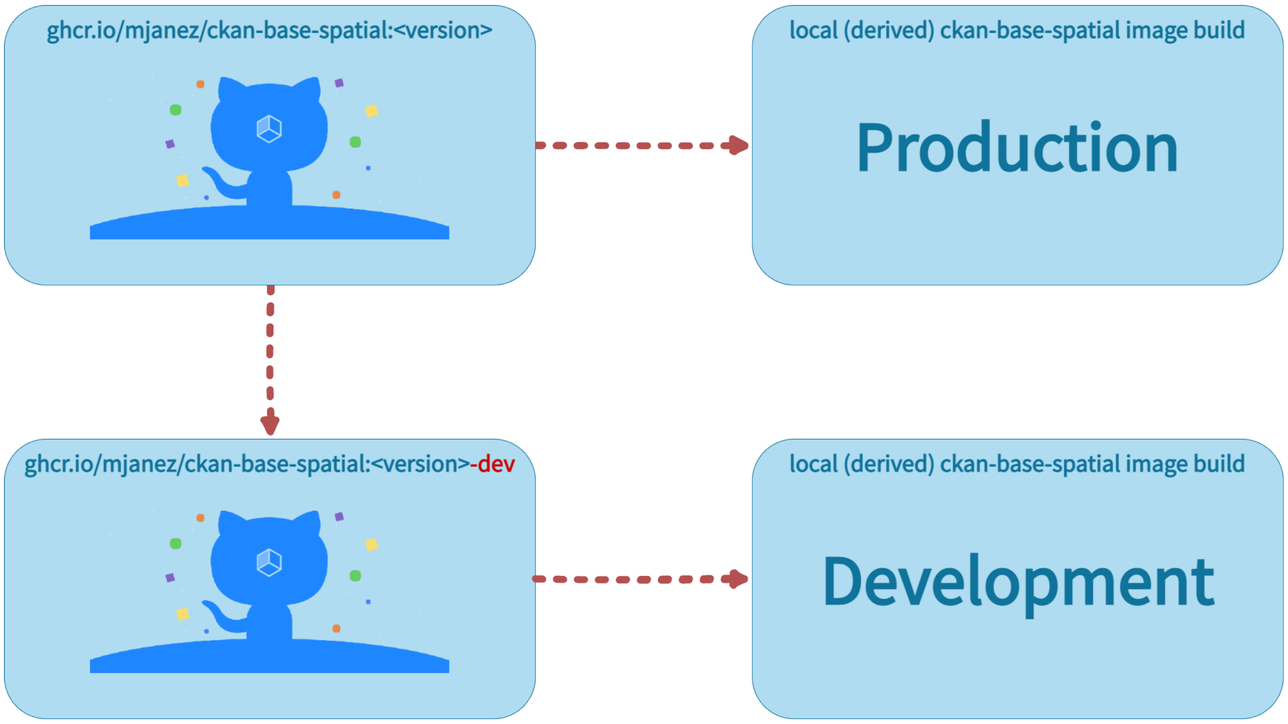
At the end of the container start sequence there should be 6 containers running.

After this step, CKAN should be running at `http://{PROXY_SERVER_NAME}/{PROXY_CKAN_LOCATION}` and ckan-pycsw at `http://{PROXY_SERVER_NAME}/{PROXY_PYCSW_LOCATION}`, i.e: http://localhost/catalog or http://localhost/csw

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
0217537f717e	ckan-docker-nginx/ ckan-docker-apache	/docker-entrypoint.…	6 minutes ago	Up 4 minutes	80/tcp,0.0.0.0:80->80/ tcp,0.0.0.0:8443->443/ tcp	ckan-docker-nginx-1/ ckan-docker-apache-1
7b06ab2e060a	ckan-docker-ckan	/srv/app/ start_ckan…	6 minutes ago	Up 5 minutes (healthy)	0.0.0.0:5000->5000/ tcp	ckan-docker-ckan-1
1b8d9789c29a	redis:7-alpine	docker-entrypoint.s…	6 minutes ago	Up 4 minutes (healthy)	6379/tcp	ckan-docker-redis-1
7f162741254d	ckan/ckan-solr:2.9-solr9-spatial	docker-entrypoint.s…	6 minutes ago	Up 4 minutes (healthy)	8983/tcp	ckan-docker-solr-1
2cdd25cea0de	ckan-docker-db	docker-entrypoint.s…	6 minutes ago	Up 4 minutes (healthy)	5432/tcp	ckan-docker-db-1
9cdj25dae6gr	ckan-docker-pycsw	docker-entrypoint.s…	6 minutes ago	Up 4 minutes (healthy)	8000/tcp	ckan-docker-pycsw-1

ckan-docker: Enhancements

CKAN images


ghcr.io/mjanez/ckan-base-spatial:<version>
local (derived) ckan-base-spatial image build
Production
ghcr.io/mjanez/ckan-base-spatial:<version>-dev
local (derived) ckan-base-spatial image build
Development

The Docker image config files used to build your CKAN project are located in the `ckan/` folder. There are two Docker files:

- `Dockerfile`: this is based on `mjanez/ckan-base-spatial:<version>`, a base image located in the Github Package Registry, that has CKAN installed along with all its dependencies, properly configured and running on uWSGI

(production setup)

- `Dockerfile.dev`: this is based on `mjanez/ckan-base-spatial:<version>-dev` also located located in the Github Package Registry, and extends `mjanez/ckan-base-spatial:<version>` to include:
    - Any extension cloned on the `./src` folder will be installed in the CKAN container when booting up Docker Compose (`docker compose up`). This includes installing any requirements listed in a `requirements.txt` (or `pip-requirements.txt`) file and running `python setup.py develop`.
    - CKAN is started running this: `/usr/bin/ckan -c /srv/app/ckan.ini run -H 0.0.0.0`.
    - Make sure to add the local plugins to the `CKAN__PLUGINS` env var in the `.env` file.
- Any custom changes to the scripts run during container start up can be made to scripts in the `setup/` directory. For instance if you wanted to change the port on which CKAN runs you would need to make changes to the Docker Compose yaml file, and the `start_ckan.sh.override` file. Then you would need to add the following line to the Dockerfile ie: `COPY setup/start_ckan.sh.override ${APP_DIR}/start_ckan.sh`. The `start_ckan.sh` file in the locally built image would override the `start_ckan.sh` file included in the base image

TIP

If you get an error like `doesn't have execute permissions`:

```
Daemon error response: failed to create shim task: OCI runtime
create failed: runc create failed: unable to start container
```

It may be necessary to give execute permissions to the file in the `Dockerfile`:

```
...
# Override start_ckan.sh
COPY setup/start_ckan.sh.override ${APP_DIR}/start_ckan.sh
RUN chmod +x ${APP_DIR}/start_ckan.sh
...
```

CKAN images enhancement

Extending the base images

You can modify the docker files to build your own customized image tailored to your project, installing any extensions and extra requirements needed. For example here is where you would update to use a different CKAN base image ie: `ckan/ckan-base-spatial:<new version>`

To perform extra initialization steps you can add scripts to your custom images and copy them to the `/docker-entrypoint.d` folder (The folder should be created for you when you build the image). Any `*.sh` and `*.py` file in that folder will be executed just after the main initialization script (`prerun.py`) is executed and just before the web server and supervisor processes are started.

For instance, consider the following custom image:

```
ckan
├── docker-entrypoint.d
│   └── setup_validation.sh
├── Dockerfile
```

We want to install an extension like ckanext-validation that needs to create database tables on startup time. We create a `setup_validation.sh` script in a `docker-entrypoint.d` folder with the necessary commands:

```
#!/bin/bash

# Create DB tables if not there
ckan -c /srv/app/ckan.ini validation init-db
```

And then in our `Dockerfile.dev` file we install the extension and copy the initialization scripts:

```
FROM ckan/ckan-base-spatial:2.9.11

RUN pip install -e git+https://github.com/frictionlessdata/ckanext-
validation.git#egg=ckanext-validation && \
    pip install -r https://raw.githubusercontent.com/frictionlessdata/
ckanext-validation/master/requirements.txt

COPY docker-entrypoint.d/* /docker-entrypoint.d/
```

TIP

There are a number of extension examples commented out in the `Dockerfile.dev` file

Applying patches

When building your project specific CKAN images (the ones defined in the `ckan/` folder), you can apply patches to CKAN core or any of the built extensions. To do so create a folder inside `ckan/patches` with the name of the package to patch (ie `ckan` or `ckanext-??`). Inside you can place patch files that will be applied when building the images. The patches will be applied in alphabetical order, so you can prefix them sequentially if necessary.

For instance, check the following example image folder:

```
ckan
├── patches
│   ├── ckan
│   │   ├── 01_datasets_per_page.patch
│   │   ├── 02_groups_per_page.patch
│   │   ├── 03_or_filters.patch
│   └── ckanext-harvest
│       └── 01_resubmit_objects.patch
├── setup
├── Dockerfile
└── Dockerfile.dev
```

INFO

Git diff is a command to output the changes between two sources inside the Git repository. The data sources can be two different branches, commits, files, etc.

- Show changes between working directory and staging area:

```
git diff > [file.patch]
```

- Shows any changes between the staging area and the repository:

```
git diff --staged [file]`
```

Applying patches in dev mode

To apply patches in development mode, you would need to follow these steps:

1. Ensure that your patches are placed in the `ckan/patches` directory. The patches should be organized into subdirectories named after the package they are intended to patch (e.g., `ckan` or `ckanext-??`). Each patch file should end with the .patch extension.

   For example, your directory structure might look like this:

   ```
   ckan
   ├── patches
   │   ├── ckan
   │   │   ├── 01_datasets_per_page.patch
   │   │   ├── 02_groups_per_page.patch
   │   │   ├── 03_or_filters.patch
   │   └── ckanext-harvest
   │       └── 01_resubmit_objects.patch
   ├── setup
   ├── Dockerfile
   └── Dockerfile.dev
   ```

2. Navigate to the `/src` directory.

3. Apply the patches using the patch command:

   ```
   find /path/to/ckan/patches -name '*.patch' -exec patch -p1 < {} \;
   ```

   This command will recursively search the `/path/to/ckan/patches` directory for files ending with `.patch` and apply them using the patch command. Replace `/path/to/ckan/patches` with the actual path to your `ckan/patches` directory.

Create an extension in development mode

You can use the ckan extension instructions to create a CKAN extension, only executing the command inside the CKAN container and setting the mounted `src/` folder as output:

```
docker compose -f docker-compose.dev.yml exec ckan-dev /bin/sh -c
"ckan -c /srv/app/ckan.ini generate extension --output-dir /srv/app/
src_extensions"
```

Then, answer the prompts to configure the plugin:

```
Extension's name [must begin 'ckanext-']: ckanext-newextension
Author's name []: Joe Bloggs
Author's email []: joe@bloggs.com
Your Github user or organization name []: joebloggs
Brief description of the project []: test creating a new extension
List of keywords (separated by spaces) [CKAN]: ckanext-newextension
Do you want to include code examples? [y/N]: y

Written: /srv/app/src_extensions/ckanext-newextension
```

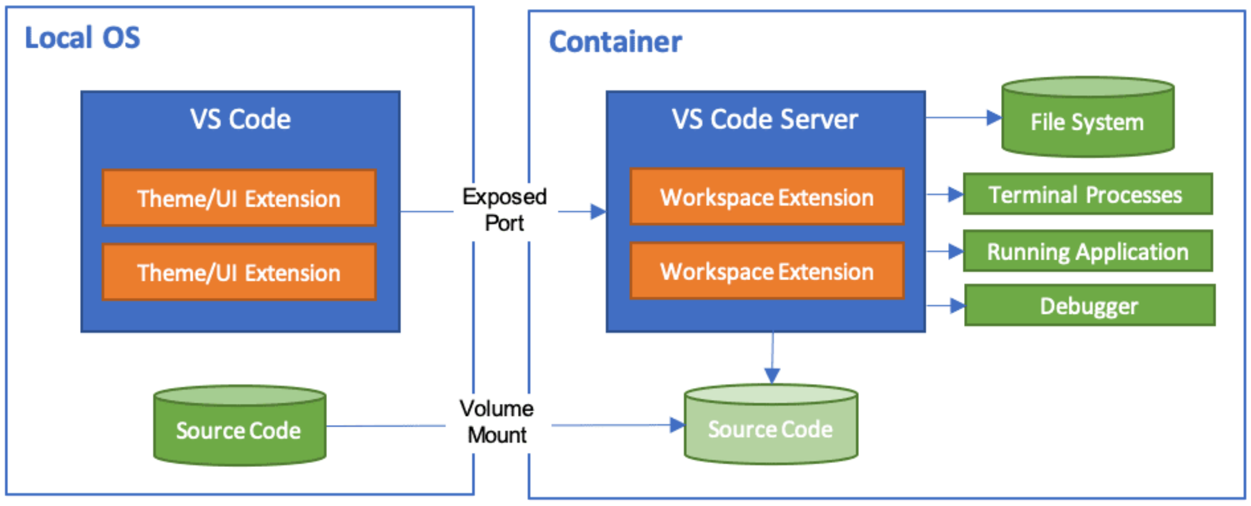
The new extension files and directories are created in the `/srv/app/src_extensions/` folder in the running container. They will also exist in the local src/ directory as local `/src` directory is mounted as `/srv/app/src_extensions/` on the ckan container. You might need to change the owner of its folder to have the appropiate permissions.

ckan-docker: Addons

Debugging

Debugging CKAN Development Instance with VSCode Dev Containers and debugpy

The Visual Studio Code Dev Containers extension is a powerful tool that enables developers to use a container as a complete development environment. With this extension, developers can open any folder inside a container and take advantage of the full range of features provided by Visual Studio Code. To do this, developers create a `devcontainer.json` file in their project that specifies how to access or create a development container with a predefined tool and runtime stack. This allows developers to work in an isolated environment, ensuring that the development environment is consistent across team members and that project dependencies are easy to manage.



To set this up:

1. Install VSCode.

2. Install the Remote Development extension for VSCode.

3. In your project directory, create a `devcontainer.json` file. This file will contain the configuration for your development container.

4. In the `devcontainer.json` file, specify the Docker image for your development container and any additional configuration settings, such as environment variables, ports to expose, and startup commands.

5. Enable `debugpy` for your development instance in your `.env` file:

```
USE_DEBUGPY_FOR_DEV=true
```

6. Start the containers in development mode and launch VS Code.

7. Install the "Dev Container" extension: press `CTRL+SHIFT+X`, type "dev container", click "install".

8. Click the `Open a Remote Window` button in the bottom-left of the VS Code window.

9. Click `Attach to Running Container...` and select your ckan-dev container, e.g. `ckan-docker-ckan-dev-1`.

10. Click the `Run and Debug` icon on the left panel then `create a launch.json`, select `Python Debugger`, `Remote Attach`, host `localhost` and port `5678`.

11. Press `F5` or click the `Run` menu and `Start Debugging`.

You can now set breakpoints and remote debug your CKAN development instance using VSCode Dev Containers and debugpy.

pdb

Add these lines to the `ckan-dev` service in the docker compose.dev.yml file

```
ports:
  - "0.0.0.0:${CKAN_PORT}:5000"

stdin_open: true
tty: true
```

Debug with pdb (example) - Interact with `docker attach $(docker container ls -qf name=ckan)`

command: `python -m pdb /usr/lib/ckan/venv/bin/ckan --config /srv/app/ckan.ini run --host 0.0.0.0 --passthrough-errors`

Reverse proxy

NGINX

The default Docker Compose configuration (`docker-compose.yml`) uses an NGINX image as the front-end (ie: reverse proxy). It includes HTTPS running on port number 8443 and an HTTP port (81). A "self-signed" SSL certificate is generated beforehand and the server certificate and key files are included. The NGINX `server_name` (ENV: `PROXY_SERVER_NAME`) directive and the `CN` field in the SSL certificate have been both set to 'localhost'. This should obviously not be used for production.

The proxy locations, ports and other NGINX options can be modified in the

`.env` file:

```
# Host Ports
NGINX_PORT_HOST=81
NGINX_SSLPORT_HOST=8443

# NGINX
NGINX_PORT=80
NGINX_SSLPORT=443
NGINX_LOG_DIR=/var/log/nginx

# Check CKAN__ROOT_PATH and CKANEXT__DCAT__BASE_URI. If you don't
need to use domain locations, it is better to use the nginx
configuration. Leave blank or use the root `/`.
PROXY_SERVER_NAME=localhost
PROXY_CKAN_LOCATION=/catalog
PROXY_PYCSW_LOCATION=/csw
```

The base Docker Compose configuration uses an NGINX image as the front-end (ie: reverse proxy). It includes HTTPS running on port number 8443. A "self-signed" SSL certificate is generated as part of the ENTRYPOINT. The ENV `PROXY_SERVER_NAME`, NGINX `server_name` directive and the `CN` field in the SSL certificate have been both set to 'localhost'. This should obviously not be used for production.

Creating the SSL cert and key files as follows: `openssl req -new -newkey rsa:4096 -days 365 -nodes -x509 -subj "/C=DE/ST=Berlin/L=Berlin/O=None/CN=localhost" -keyout ckan-local.key -out ckan-local.crt` The `ckan-local.*` files will then need to be moved into the nginx/setup/ directory

Apache HTTP Server

The Docker Compose configuration (`docker-compose.apache.yml`) uses an httpd image as the front-end. It has two routes for the ckan (default location:

`/catalog`) and ckan-pycsw (default location: `/csw`) services.

The proxy locations, ports and other Apache Web Server options can be modified in the `.env` file:

```
# Host Ports
APACHE_PORT_HOST=81

# Apache HTTP Server
APACHE_VERSION=2.4-alpine
APACHE_PORT=80
APACHE_LOG_DIR=/var/log/apache

# Check CKAN__ROOT_PATH and CKANEXT__DCAT__BASE_URI. If you don't
need to use domain locations, it is better to use the nginx
configuration. Leave blank or use the root `/`.
PROXY_SERVER_NAME=localhost
PROXY_CKAN_LOCATION=/catalog
PROXY_PYCSW_LOCATION=/csw
```

envvars

The ckanext-envvars extension is used in the CKAN Docker base repo to build the base images. This extension checks for environmental variables conforming to an expected format and updates the corresponding CKAN config settings with its value.

For the extension to correctly identify which env var keys map to the format used for the config object, env var keys should be formatted in the following way:

All uppercase
Replace periods ('.') with two underscores ('__')

Keys must begin with 'CKAN' or 'CKANEXT', if they do not you can prepend them with '`CKAN___`'

For example:

- `CKAN__PLUGINS="envvars image_view text_view recline_view datastore datapusher"`
- `CKAN__DATAPUSHER__CALLBACK_URL_BASE=http://ckan:5000`
- `CKAN___BEAKER__SESSION__SECRET=CHANGE_ME`

These parameters can be added to the `.env` file

For more information please see ckanext-envvars

CAUTION

When deploying under a proxy, such as in a corporate environment, to avoid errors when resolving urls with container_names/hostnames associated with the container on internal networks, use the `no_proxy` variable, in lower case, with the names of the services/containers, the IP of the Docker network, etc.

e.g:

```
no_proxy="127.0.0.1,192.168.192.0/
23,172.0.0.0/0,redis,solr,${DB_CONTAINER_NAME}"`
```

Datastore

The Datastore database and user is created as part of the entrypoint scripts for the db container.

xloader

This deployment replaces DataPusher with XLoader using Supervisor, more info about other alternatives on the wiki page for this: https://github.com/ckan/ckan-docker/wiki/Replacing-DataPusher-with-XLoader

ckan-pycsw

ckan-pycsw is a docker compose environment (based on pycsw) for development and testing with CKAN Open Data portals.

Available components:

- **pycsw**: The pycsw app. An OARec and OGC CSW server implementation written in Python.
- **ckan2pycsw**: Software to achieve interoperability with the open data portals based on CKAN. To do this, ckan2pycsw reads data from an instance using the CKAN API, generates ISO-19115/ISO-19139 metadata using pygeometa, or a custom schema that is based on a customized CKAN schema, and populates a pycsw instance that exposes the metadata using CSW and OAI-PMH.

Harvester consumers on a deployed CKAN

ckanext-harvest supervisor allows you to harvest metadata from multiple sources on a production deployment. Here it is deployed by a worker consumers in the `ckan` container, also the `ckanext-harvest` extension and other custom harvesters (`ckanext-schemingdcat` or `ckanext-dcat`) are included in the CKAN docker images.

TIP

To enable harvesters you need to set up in the `.env` file the `CKAN__PLUGINS` variable with the `harvest` plugin: https://github.com/mjanez/ckan-docker/blob/a18e0c80d9f16b6d9b6471e3148d48fcb83712bd/.env.example#L126-L127

ckan-docker: Tips

CKAN. Backups

PostgreSQL offers the command line tools `pg_dump` and `pg_restore` for dumping and restoring a database and its content to/from a file.

Backup service for db container

1. Create a new file called `ckan_backup_custom.sh` and open it in your preferred text editor.

2. Add the following code to the script, replacing the placeholders with your actual values:

```
#!/bin/bash

# Set the necessary variables
CONTAINER_NAME="db"
DATABASE_NAME="ckandb"
POSTGRES_USER="postgres"
POSTGRES_PASSWORD="your_postgres_password"
```

3. Replace the following placeholders with your actual values:

    - `your_postgres_password`: The password for the PostgreSQL user.
    - `/path/to/your/backup/directory`: The path to the directory where you want to store the backup files.

    > **WARNING**
    >
    > If you have changed the values of the PostgreSQL container, database or user, change them too. Check that `zip` package is installed: `sudo apt-get install zip`

4. Save and close the file.

5. Make the script executable:

    ```
    chmod +x ckan_backup_custom.sh
    ```

6. Open the `crontab` for the current user:

    ```
    crontab -e
    ```

7. Add the following line to schedule the backup to run daily at midnight (adjust the schedule as needed):

    ```
    0 0 * * * /path/to/your/script/ckan_backup_custom.sh
    ```

    > **NOTE**
    >
    > Replace `/path/to/your/script` with the actual path to the

> `ckan_backup_custom.sh` script.

8. Save and close the file.

The cronjob is now set up and will backup your CKAN PostgreSQL database daily at midnight using the custom format. The backups will be stored in the specified directory with the timestamp in the filename.

> **NOTE**
>
> Sample scripts for backing up CKAN: `doc/scripts`

Restore a backup

If need to use a backup, restore it:

1. First clean the database. **Caution, this will delete all data from your CKAN database!**

```
docker exec -it ckan /bin/bash -c "export TERM=xterm; exec bash"

# Delete everything in the CKAN database, including the tables,
to start from scratch
ckan -c $CKAN_INI db clean
```

2. After cleaning the database you must do either initialize it or import a previously created dump.

```
docker exec -i -e PGPASSWORD=$POSTGRES_PASSWORD
$POSTGRESQL_CONTAINER_NAME pg_restore -U $POSTGRES_USER --clean --
if-exists -d $DATABASE_NAME < /path/to/your/backup/directory/
ckan.dump
```

3. Restart the `ckan` container.

CKAN. Manage new users

- Create a new user from the Docker host, for example to create a new user called `user_example`

  ```
  docker exec -it <container-id> ckan -c ckan.ini user add
  user_example email=user_example@localhost

  # Admin user
  docker exec -it <container-id> ckan -c ckan.ini sysadmin add
  admin_example email=admin_example@localhost name=admin_example
  ```

  To delete the 'user_example' user

  ```
  docker exec -it <container-id> ckan -c ckan.ini user remove
  user_example`
  ```

- Create a new user from within the ckan container. You will need to get a session on the running container

  ```
  ckan -c ckan.ini user add user_example
  email=user_example@localhost`
  ```

  To delete the 'user_example' user

  ```
  ckan -c ckan.ini user remove user_example`
  ```

Docker Compose. Configure a docker compose

service to start on boot

To have Docker Compose run automatically when you reboot a machine, you can follow the steps below:

1. Create a systemd service file for Docker Compose. You can create a file named `ckan-docker-compose.service` in the `/etc/systemd/system/` folder with the following content:

   ```
   [Unit]
   Description=CKAN Docker Compose Application Service
   Requires=docker.service
   After=docker.service

   [Service]
   User=docker
   Group=docker
   Type=oneshot
   RemainAfterExit=yes
   WorkingDirectory=/path/to/project/ckan-docker/
   ExecStart=/bin/docker compose up -d
   ExecStop=/bin/docker compose down
   TimeoutStartSec=0

   [Install]
   WantedBy=multi-user.target
   ```

2. Replace `/path/to/project/ckan-docker/` with the path where your project's `docker-compose.yml` file is located and and check the path to the docker compose binary on execution and stop: `/bin/docker`. Also change the `User` / `Group` to execute the service.

3. Load the systemd service file with the following command:

```
sudo systemctl daemon-reload
```

4. Enables the service to start automatically when the machine boots up:

```
sudo systemctl enable ckan-docker-compose
```

5. You can now start the service with the following command:

```
sudo systemctl start ckan-docker-compose
```

6. If you want to stop or check the status of the service, use the following commands:

```
# Stop the service
sudo systemctl stop ckan-docker-compose

# Check the status
sudo systemctl status ckan-docker-compose
```

Ansible installation

Overview

CKAN is a powerful open data platform that provides a suite of tools to facilitate the publishing, sharing, finding and using of data. This document provides a step-by-step guide to using an Ansible playbook for the deployment of a custom CKAN for spatial data management in different environments.

ckan-ansible

`ckan-ansible` is a custom Ansible playbook for deploying CKAN in different environments. The playbook is designed to automate the installation and configuration of CKAN, PostgreSQL, Solr, Redis, and NGINX. The playbook is designed to be flexible and customizable, allowing you to configure the deployment to suit your specific requirements.

Deployments available for the following OS:

- RedHat Enterprise Linux 9 (RHEL 9)
- RedHat Enterprise Linux 8 (RHEL 8)
- [WIP] Debian 12 (Bookworm)
- [WIP] Ubuntu 20.04 (Focal Fossa)

WARNING

This is a **custom installation** with specific extensions for spatial data and metadata GeoDCAT-AP/INSPIRE profiles. For official installations see CKAN Documentation: Installation.

Available components:

CKAN is a world-leading open source data portal platform that provides a suite of tools to streamline the publishing, sharing, finding, and using of data. 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.

The non-CKAN componentes are as follows:

- PostgreSQL: A powerful, open source object-relational database system. CKAN uses PostgreSQL to store its data. In a CKAN installation, you would need to set up a PostgreSQL database and configure CKAN to use it.
- Solr: A popular, open source search platform from the Apache Lucene project. CKAN uses Solr as its search engine. When you create or update datasets, resources, or other objects in CKAN, it updates the Solr index. Then, when you search in CKAN, it queries Solr and shows the results. The Solr index data is stored in a named volume, similar to the PostgreSQL data. In this deployment Solr has a custom spatial schema that allows for geographic searches.
- Redis: An open source, in-memory data structure store, used as a database, cache, and message broker. CKAN uses Redis as a message broker for its background jobs. When CKAN needs to perform a task that may take a long time, such as updating the search index for a large number of datasets, it adds a job to the Redis queue, which can then be processed in the background.
- NGINX: A free, open-source, high-performance HTTP server and reverse proxy. CKAN uses NGINX as a reverse proxy to route incoming HTTP requests to the CKAN application. NGINX can also serve static files, handle

SSL encryption, and load balance requests between multiple CKAN instances. The NGINX configuration for CKAN includes both SSL and non-SSL endpoints, allowing for secure communication over HTTPS.

Requirements

Prerequisites

Before you can use `ckan-ansible`, you need to install:

- Ansible

CKAN: Installation and configuration

CKAN Ansible Deployment

Clone this repository to your local machine and edit the variables of the `playbook/host_vars/production_01.yml`:

```
git clone https://github.com/mjanez/ckan-ansible.git && cd ckan-
ansible
```

PLAYBOOK CONFIG

Remember to change the CKAN configuration variables before running the Ansible playbook. Specifically the host user/pwd info (`ansible_user`, `ansible_password`, etc.) and CKAN configuration: `ckan_sysadmin_name`, `ckan_sysadmin_password` and `ckan_sysadmin_email`. Also the `proxy_server_name` and `nginx_port` for correct deployment.

Edit the `inventory` folder hosts vars and add the target deployment servers IP addresses or `hostname` for the specific environment.

Customize the deployment configurations in `host_vars/*` to match your requirements. Modify any necessary variables such as database credentials, CKAN versions, and other specific settings.

SSH AUTHENTICATION

Also if using a SSH password authentication for private repos create a SSH key pair and copy the keys to the `./playbook/roles/common/files/keys`. The filenames of the keypair files must begin with id_ (e.g. `id_rsa` + `id_rsa.pub`)

Sample deployment

1. Select the environment you want to deploy, e.g: `rhel`.

2. Edit the `playbook/host_vars/production_01.yml`. Put the path to the SSH private key if is not using password authentication (`ansible_ssh_private_key_file`/`ansible_ssh_pass`).

3. Run the Ansible playbook to deploy CKAN on the target server. The following command will deploy CKAN on the target server using the playbook configuration. The `-vvv` flag is used for verbose output:

```
# Location of the ansible.cfg file based on the clone directory
export ANSIBLE_CONFIG=$(pwd)/playbook/ansible.cfg

# Location if ckan-ansible is cloned in the home directory
export ANSIBLE_CONFIG=$HOME/ckan-ansible/playbook/ansible.cfg

# Run the ansible playbook, Verbose with  -vvv
ansible-playbook $HOME/ckan-ansible/playbook/playbook.yml
```

The `ANSIBLE_CONFIG` environment variable is used to specify the location of the `ansible.cfg` file. This is useful when you have multiple Ansible configurations and you want to specify which one to use, eg. `rhel-9`, `ubuntu-20.04`, etc.

HOST_VARS

> The `*/host_vars/*.yml` file contain customizable configuration variables for deployment, including database credentials, CKAN version, and web server configuration. Review and modify these before running the Ansible playbook.

Check the services are running

After the deployment, you can check the status of the services using the `supervisorctl` command. This command provides a command-line interface to the Supervisor process control system, which allows you to control and monitor your services.

The services generated by ckan-Ansible include:

- `ckan`, an open-source DMS (data management system) for powering data hubs and data portals which is served via uWSGI and NGINX.
- `ckan-pycsw`, a full-featured web service for cataloging geospatial data. Software to achieve interoperability with the open data portals based on CKAN. To do this, `ckan2pycsw` reads data from an instance using the CKAN API, generates INSPIRE ISO-19115/ISO-19139 [^3] metadata using pygeometa, or another custom schema, and populates a pycsw instance that exposes the metadata using CSW and OAI-PMH.
- `ckan-xloader`, worker for quickly load data into DataStore. A replacement for DataPusher.
- Workers used for remote harvesting. The `ckan_harvester_run` worker is used to periodically run, the harvesters and the `ckan_harvester_fetch` worker is used to retrieves the data from the remote servers and prepares it for the `ckan_harvester_run` worker. Also the `ckan_harvester_gather`

worker is used to identifying the remote resources that need to be harvested. Finally, the `ckan_harvester_clean_log` worker is used to periodically clean the logs of the harvesters.

To check the status of these services, you can use the `supervisorctl status` command. Here's an example:

```
$ supervisorctl status
ckan_fetch_consumer                RUNNING   pid 2684195, uptime 0:00:50
ckan_gather_consumer               RUNNING   pid 2684193, uptime 0:00:50
ckan_harvester_clean_log           STOPPED   Not started
ckan_harvester_run                 EXITED    May 07 01:12 PM
ckan_pycsw                         RUNNING   pid 2684197, uptime 0:00:50
ckan_uwsgi:ckan_uwsgi-00           RUNNING   pid 2684194, uptime 0:00:50
ckan_xloader:ckan_xloader-00       RUNNING   pid 2684198, uptime 0:00:50
```

CKAN API: Basics

INFO

`params`: Parameters to pass to the action function. The parameters are specific to each action function.

- `fl` (text): Fields of the dataset to return. The parameter controls which fields are returned in the solr query. `fl` can be `None` or a list of result fields, such as: `id,name,extras_custom_schema_field`.

  Example All datasets with the fields `id`, `name`, `title` and a custom schema field `extras_inspire_id`:

  ```
  {ckan-instance}/api/3/action/
  package_search?fl=id,name,title,extras_inspire_id
  ```

- `fq` (text): Any filter queries to apply.

  Example All datasets that have tag `economy`:

  ```
  http://demo.ckan.org/api/3/action/package_search?fq=tags:economy
  ```

- `rows` (int): The maximum number of matching rows (datasets) to return. (optional, default: `10`, upper limit: `1000` unless set in site’s configuration `ckan.search.rows_max`)

More info: CKAN API Documentation and data.gov.uk

List datasets by fields

Request: `{ckan-instance}/api/3/action/package_search?fl=id,extras_publisher_name`

Response:

```
{
  "help": "{ckan-instance}/api/3/action/help_show?name=package_search",
  "success": true,
  "result": {
    "count": 32,
    "facets": {},
    "results": [
      {
        "id": "e4a607d0-0875-4043-b8c7-36f731ba5ca8",
        "publisher_name": "Example publisher"
      },
      {
        "id": "5319a6b3-f439-4f53-9732-71699b9f62c8",
        "publisher_name": "Example publisher"
      },
      {
        "id": "02a30269-7665-4f6a-a43d-c288003f5cbb",
        "publisher_name": "Example publisher"
      }
    ],
    "sort": "score desc, metadata_modified desc",
    "search_facets": {}
  }
}
```

All datasets in organization (with some fields)

Request: `{ckan-instance}/api/3/action/package_search?fq=organization:iepnb&fl=id,name,extras_alternate_identifier&rows=100`

Response:

```
{
```

All info about a dataset by field

Request: `{ckan-instance}/api/3/action/package_search?q=name:"spa_example_dataset_1_2023"`

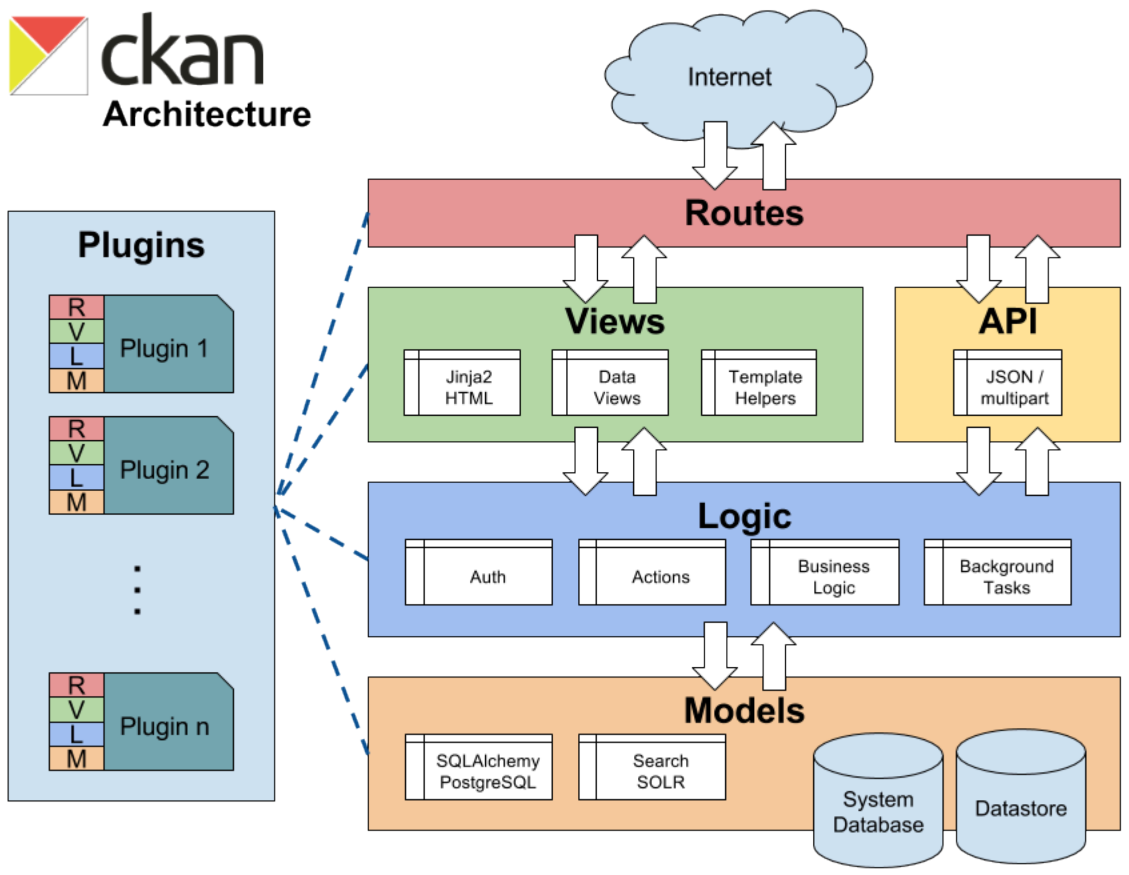
Response:

```
{
  "help": "https://demo.ckan.org/api/3/action/help_show?name=package_search",
  "success": true,
  "result": {
    "count": 1,
    "facets": {},
    "results": [
      {
        "author": "Test Author",
        "author_email": "test@email.com",
        "creator_user_id": "47c7f1b1-0ef5-4d7b-b43c-811c51c9e349",
        "id": "c322307a-b871-44fe-a602-32ee8437ff04",
        "isopen": true,
        "license_id": "cc-by",
        "license_title": "Creative Commons Attribution",
        "license_url": "http://www.opendefinition.org/licenses/cc-by",
        "maintainer": "Test Maintainer",
        "maintainer_email": "test@email.com",
        "metadata_created": "2021-04-09T11:39:37.657233",
        "metadata_modified": "2022-05-20T09:20:43.998956",
        "name": "sample-dataset-1",
        "notes": "A CKAN Dataset is a collection of data resources (such as
files), together with a description and other information (what is known as
metadata), at a fixed URL. \r\n\r\n",
        "num_resources": 9,
        "num_tags": 8,
        "organization": {
          "id": "1fa89238-ee96-4439-a885-22d15244d070",
          "name": "sample-organization",
          "title": "Sample Organization",
          "type": "organization",
          "description": "This is a sample organization.",
          "image_url": "2022-05-20-084702.929838siurana.jpg",
          "created": "2021-04-09T14:27:17.753798",
          "is_organization": true,
          "approval_status": "approved",
```

CKAN: Development

CKAN code architecture

CKAN's code architecture is designed with specific coding standards in mind to maintain consistency with its intended design and architecture.



TIP

More info at: CKAN Code Architecture

Blueprints

CKAN is based on Flask and uses Blueprints. Default blueprints are defined along views in `ckan.views` and extended with the `ckan.plugins.interfaces.IBlueprint` plugin interface.

Views

Views process requests by reading and updating data with action functions and return a response by rendering Jinja2 templates. CKAN views are defined in `ckan.views` and templates in `ckan.templates`.

Template Helper Functions

Template helper functions are used for code that is reused frequently or code that is too complicated to be included in the templates themselves. They should never perform expensive queries or update data.

Action Functions

Whenever some code 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.

Logic

Logic includes action functions, auth functions, background tasks and business logic. Action functions have a uniform interface accepting a dictionary of simple strings lists, dictionaries or files. They return simple dictionaries or raise one of a small number of exceptions.

Models

Ideally SQLAlchemy should only be used within `ckan.model` and not from other packages such as `ckan.logic`.

Deprecation

Anything that may be used by extensions, themes or API clients needs to maintain backward compatibility at call-site. To deprecate a function use the `ckan.lib.maintain.deprecated` decorator and add "deprecated" to the function's docstring.

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:

- Translating CKAN
- Testing CKAN
- Writing documentation
- CKAN code architecture
- CSS coding standards
- HTML coding standards
- JavaScript coding standards
- Python coding standards
- String internationalization
- Unicode handling

- Testing coding standards
- Frontend development guidelines
- Database migrations
- Upgrading CKAN’s dependencies

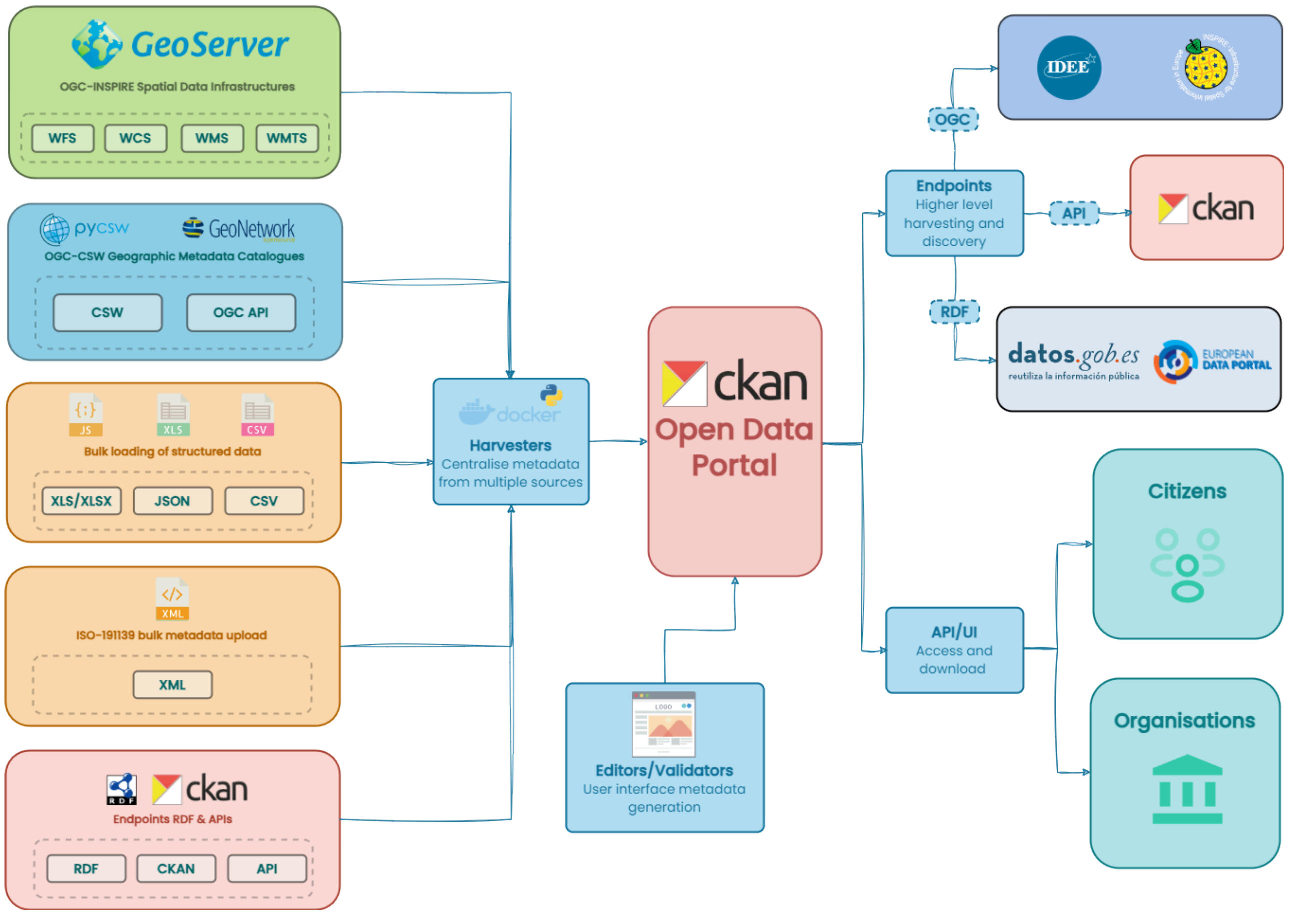
CKAN Harvesters

`ckanext-harvest`, is a powerful extension for CKAN, a leading open-source data portal platform. This extension provides a common harvesting framework that allows CKAN extensions to gather data from various sources and bring it into the CKAN instance.

The harvesting process involves two main steps: gathering and fetching. The gather stage identifies all the objects (datasets, files, etc.) that need to be harvested from the source, while the fetch stage retrieves the identified objects.

CKAN Harvester supports a variety of harvesting sources and formats, and it can be extended to support more. It also provides both a Command Line Interface (CLI) and a Web User Interface (WUI) to manage harvesting sources and jobs, making it a versatile tool for data collection and management.

The extension requires CKAN v2.0 or later and can use either Redis or RabbitMQ as its backend. It also offers a range of configuration options to customize the harvesting process according to your needs.


GeoServer
OGC-INSPIRE Spatial Data Infrastructures
WFS
WCS
WMS
WMTS
pycsw
GeoNetwork
OGC-CSW Geographic Metadata Catalogues
CSW
OGC API
JS
XLS
CSV
Bulk loading of structured data
XLS/XLSX
JSON
CSV
XML
ISO-19139 bulk metadata upload
XML
RDF
ckan
Endpoints RDF & APIs
RDF
CKAN
API
docker
Harvesters
Centralise metadata from multiple sources
ckan
Open Data Portal
Editors/Validators
User interface metadata generation
OGC
Endpoints
Higher level harvesting and discovery
API
RDF
IDEE
ckan
datos.gob.es
reutiliza la información pública
EUROPEAN DATA PORTAL
API/UI
Access and download
Citizens
Organisations

Custom harvester

Overview

A Harvester in the context of CKAN, is a component or process that is responsible for sourcing, collecting, and importing data from various external data sources into the CKAN instance.

The Harvester operates in two main stages: the gather stage and the fetch stage. In the gather stage, the Harvester identifies all the objects (like datasets or files) that need to be harvested from the source. In the fetch stage, the Harvester retrieves the identified objects and brings them into the CKAN instance.

Harvesters can be configured to work with a variety of data sources and formats, making them a versatile tool for populating a CKAN instance with data. They can be used to automate the process of data collection, reducing the need for manual data entry and ensuring that the CKAN instance is regularly updated with fresh data from the source.

In addition to sourcing and importing data, Harvesters also provide features like error notifications, job timeout settings, and the ability to avoid overwriting certain fields, offering a comprehensive solution for data harvesting needs in CKAN.

Develop a harvester

To develop a new harvester, you'll need to follow these steps:

1. Create a new Python file in your extension `ckanext/myplugin/harvesters/` directory. The name of the file should be descriptive of the type of data source that the harvester will process. For example, if you're creating a harvester to extract data from a REST API, you might name the file `rest_api_harvester.py`.

2. In this file, you'll need to define a new class that inherits from `ckanext.harvest.harvesters.base.HarvesterBase`. This class should implement at least the following methods:

    - `info(self)`: This method should return a dictionary with information about the harvester. It should include at least the keys 'name', 'title', and 'description'.

    - `gather_stage(self, harvest_job)`: This method is responsible for gathering the identifiers of the objects to be harvested and should return a list of these identifiers.

  - `fetch_stage(self, harvest_object)`: This method receives a harvest object (identified by one of the identifiers returned in the `gather_stage` method) and should return a dictionary with the object's data.
  - `import_stage(self, harvest_object)`: This method receives a harvest object (with the data already extracted in the `fetch_stage` method) and should create or update the corresponding dataset in CKAN.

Here's an example of what a basic harvester might look like:

```
from ckanext.harvest.harvesters.base import HarvesterBase

class RestAPIHarvester(HarvesterBase):
    def info(self):
        return {
            'name': 'rest_api',
            'title': 'REST API Harvester',
            'description': 'A harvester for REST APIs'
        }

    def gather_stage(self, harvest_job):
        # Code to gather the identifiers of the objects to be
harvested from the REST API would go here.
        return []

    def fetch_stage(self, harvest_object):
        # Code to extract the data of the object from the REST API
would go here.
        return True

    def import_stage(self, harvest_object):
        # Code to create or update the dataset in CKAN would go here.
        return True
```

3. Once you've defined your harvester class, you need to register it so that CKAN recognizes it. This is done in the `plugin.py` file in the `ckanext/harvest/` directory. Here, you should import your harvester class and add it to the `harvesters` list in the `after_map` method of the `HarvestPlugin` class.

```
from ckanext.harvest.harvesters import HarvesterBase,
rest_api_harvester

class HarvestPlugin(plugins.SingletonPlugin):
    plugins.implements(plugins.IConfigurer)
    plugins.implements(plugins.IRoutes, inherit=True)

    harvesters = []

    def after_map(self, map):
        self._register_harvesters()
        return map

    def _register_harvesters(self):
        self.harvesters.append(rest_api_harvester.RestAPIHarvester())
```

4. Finally, you need to restart CKAN for the changes to take effect.

CAUTION

Please note that this is a very basic example. Depending on the data source, you might need to implement additional logic in the `gather_stage`, `fetch_stage`, and `import_stage` methods. Additionally, you might need to implement other methods, such as `validate_config` (to validate the harvester's configuration) or `get_original_url` (to get the original URL of the harvested objects).