# Creating a harvester using the CKAN data store API

## Table of Contents



## Versions

| | | |
|---|---|---|
| 1.0 | BV | 2021-02-08 |
| Initial version | | |
| 2.0 | BV | 2021-11-17 |
| Major rework for workshop Gemeente Almere | | |

## Introduction

There are different methods for harvesting data of third party API's and insert them in a Dataplatform/Datacatalogus time series database. What is the best method depends a) on the use case at hand and b) the resources available. This chapter describes how to create a harvester using the CKAN data store API. Other options include using for instance FME to harvest third party API's.

To be able to create a harvester using the CKAN datastore API you will need an API key with sufficient privileges for your organization. While personal API keys can be used, creating an automation user with an automation API key is a better option.

Creating a harvester entails different steps: in the preparation phase you should download example data, create a package and a resource in CKAN and upload the Styled Layer Descriptor to CKAN if you want to publish the data using GeoServer. Subsequently, the actual harvester can be implemented. In this document, Python will be used to implement the harvester, but other programming languages can be used as well of course. Once the harvester has been implemented, it can be deployed to a server where it can run reliably.

The document is intended for IT staff who are familiar with programming and CKAN.

## Download example data

The first step consists of downloading example data from the third party data API. These example data allow you to come up with a strategy to harvest the data and to decide on how to store these data in a time series database. In this example we have downloaded a file contain road construction statuses from the “Nationaal Dataportaal Wegverkeer” open data portal. They provide XML files which have been compressed. See Figure 1 for an example.


*Figure 1 Example from NDW data*

After decompression the data can be inspected and it can be concluded that the XML contains “situation” nodes of which we want to store certain aspects in our database, for instance the severity, the location and the applicable time period.

## Create a package using either the CKAN user interface or the CKAN API

CKAN organizes data using organizations, packages and resources. You should be a member of an organization to be able to create a package. The package can either be created manually or via an API call. See Figure 2 for an example of a package created by hand. To create a package a number of mandatory metadata fields must be provided. Optional metadata fields can be provided. The package

also comes with a number of fields which allow you to control whether or not the data is being published using GeoServer and if the metadata is being forwarded to data.overheid.nl and Nationaal Georegister.

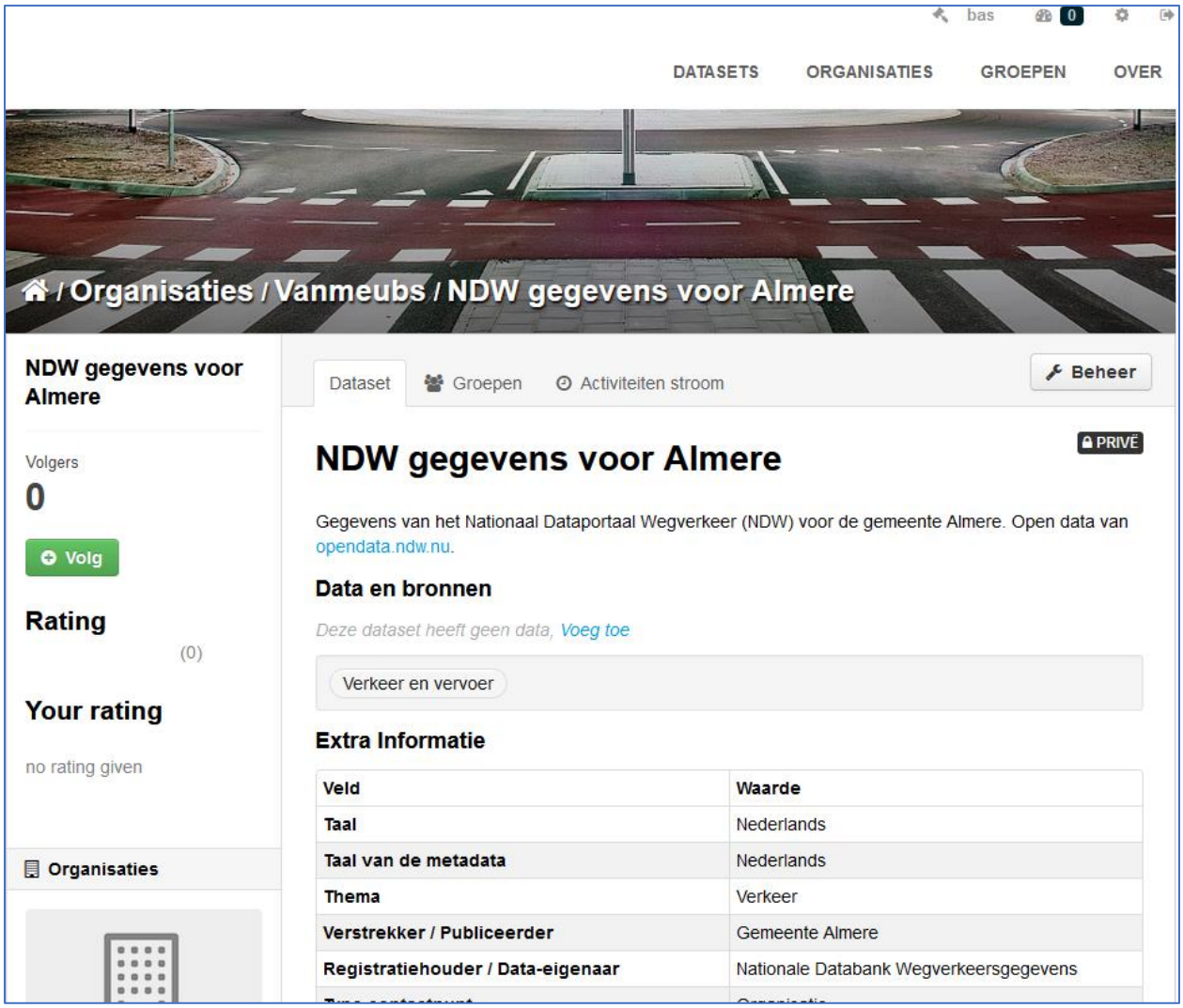


*Figure 2 Package in CKAN*

## Upload the SLD resource (optional)

If you want to publish the resource using GeoServer with a specific classification and symbolization, you should first upload the Styled Layer Descriptor (SLD) you want to apply. SLD's are XML documents describing the classification and symbolization for a feature type and can be created using for instance QGIS. Provide a name and a description for the SLD. Format should be "SLD". The other metadata fields for the SLD are optional. See Figure 3 for an example of an SLD resource. The SLD resource can be created manually or via an API call. The SLD should be created before the resource containing the actual data. If the SLD is created after this resource, an update of the resource with the actual data should be triggered to trigger application of the SLD to the resource. The SLD should be a valid XML. DataPlatform/DataCatalog do not validate this though. Validating the XML would introduce the risk of creating an XXE vulnerability.

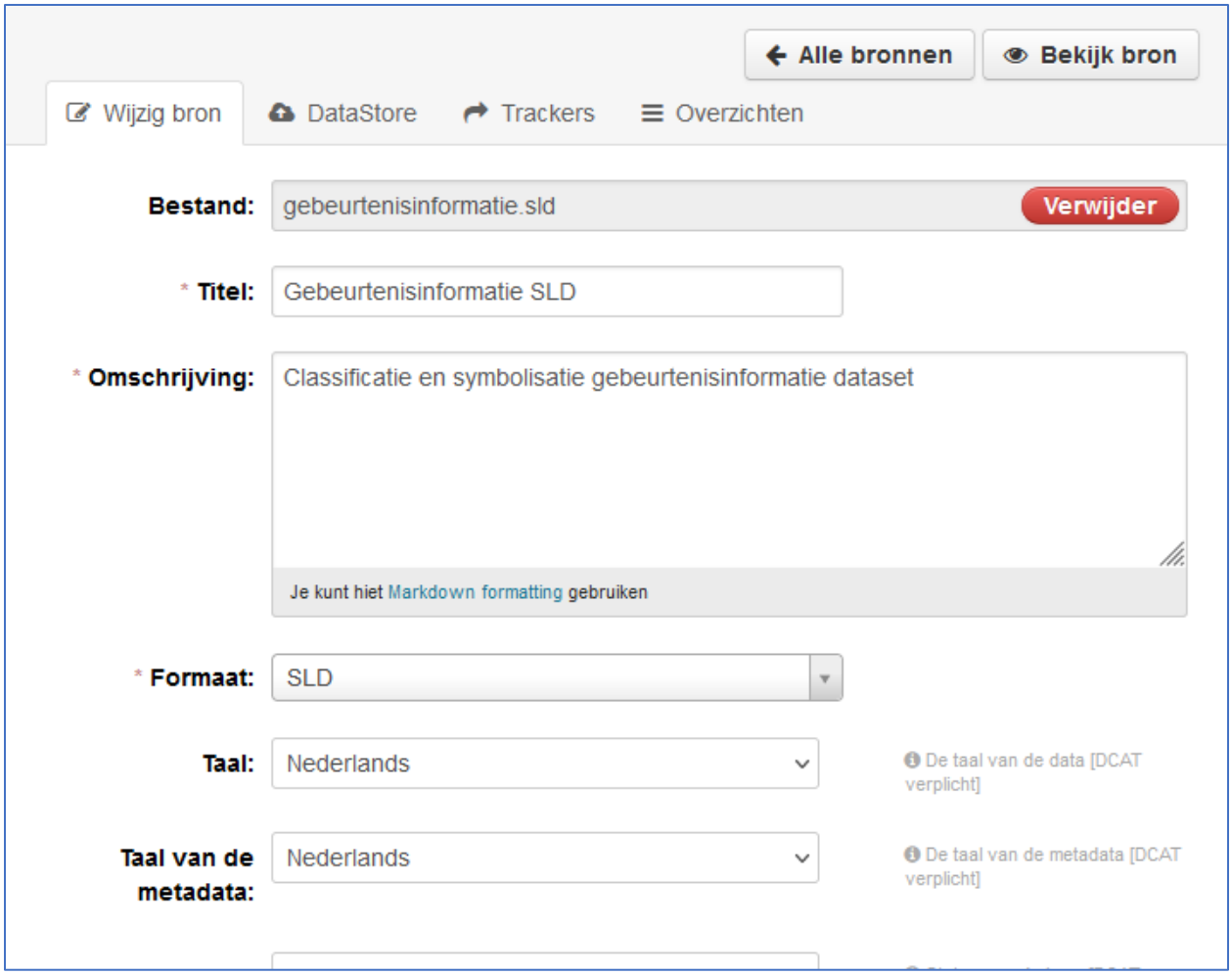


*Figure 3 Example of SLD resource*

## Create a resource using the CKAN API

Resources can also be created manually or via an API call. Creating the resource using the CKAN API has the advantage that it allows you to choose your own resource identifier which allows you to a) choose a readable resource identifier and b) use the same identifier on test, acceptance and production. This reduces the amount of differences between running the harvester on test, acceptance and production. See below for an example of the API call to create a resource.

```
{
    "fields": [
        {
            "id": "situation_id",
            "type": "text"
        }, {
            "id": "version",
            "type": "text"
        }, {
            "id": "severity",
            "type": "text"
        }, {
```

```
            "id": "causes",
            "type": "text"
        }, {
            "id": "geom",
            "type": "text"
        }
    ],
    "force": "true",
    "primary_key": [
        "situation_id"
    ],
    "resource": {
        "id": "gebeurtenisinformatie",
        "name": "Gebeurtenisinformatie",
        "description": "Gebeurtenisinformatie van het NDW. ",
        "license_id": "notspecified",
        "package_id": "gebeurtenisinformatie",
        "layer_srid": "4326"
    },
    "records": [
        {
            "situation_id": "just_testing_high",
            "version": "1",
            "severity": "high",
            "causes": "Gaten in de weg repareren, hoge prioriteit",
            "geom": "POINT(5.0 52.0)"

    ]
}
```

The API call is sent to <ckan url>/api/3/action/datastore_create and consists of a number of elements:

- The “fields” array contains the fields that should be created. To create a spatial column, either a field with type text and a name starting with “geom” should be created (in which geometries in Well Known Text format should be inserted) or two columns with type float and names starting with “lon” and “lat” should be created. Upon creation of the resource in the datastore, a spatial column will be added if these are present. This column will either be named “the_geom_from_wkt” or the “the_geom_from_lon_lat”, depending on which of the two methods was used. It will be used to publish the layer in GeoServer;
- The “primary key” array contains which fields should be used as primary key. This is important to control what happens is a record is upserted: if just the entity ID is used as primary, this entity will be updated constantly, and as a consequence to current status of the entity will always be presented. If the entity ID plus a timestamp is used as primary key, new records are being added all the time and a time series database is created.
- The resource object contains some metadata for the resource, amongst others under which package the resource must be created and (in case of a resource with spatial data) the EPSG code of the coordinate reference system of the coordinates;
- The records object contains an example record. It is needed when creating the resource and can be deleted later sending the API call below to <ckan url>/api/3/action/datastore_delete.

```
{
```

```
    "resource_id": "gebeurtenisinformatie",
    "force": true,
    "filters": {
        "situation_id": "just_testing_low"
    }
}
```

If the data should be published using GeoServer, a layer will be created in GeoServer. The layer will be in a workspace which includes the name of the CKAN instance (which is the same as part of the URL, the bit before ".dataplatform.nl" but then with underscores _ instead of dashes -. For example: the layers for CKAN https://tst-ckan-dataplatform-nl.dataplatform.nl/ reside in the GeoServer workspace workspace_datastore_ckan_dataplatform_nl. The GetCapabilities document can be downloaded at https://<ckan_url>/geoserver/workspace_datastore_ckan_dataplatform_nl/ows?. The name of the layer is identical to the "ckan_" plus the resource ID. The "ckan_" prefix is added to accommodate for resources identifiers which start with a number, this is not supported in all components.

## Implement the actual harvester

An example harvester (including the Data store API calls discussed in the previous section) can be found in the Civity Github repository at https://github.com/CivityNL/Workshop/tree/main/diy_data_harvester.

Since we will be creating the harvester using Python, it is advisable to create a Python virtual environment in which the dependencies needed for the harvester can be installed without interfering with other software components which might be using Python. Alternatively, a Docker Python image can be used in which the dependencies can be installed with the same effect. The harvester should perform the following tasks:

1. Download the data from the 3rd arty API offering the data. If the 3rd party API offers the data in different formats, use the format which is the easiest to process;
2. Parse the data using preferably a standard JSON/XML/CSV library;
3. Decide if a record has to be inserted, for instance by checking if the location of the record intersects with a polygon of the area of interest or another criterium;
4. Upsert the record in CKAN using a Data store API call (see below). In case of a dataset with a spatial column, the geometry should be inserted twice: once in the geom or lon/lat column(s) which have been created in the Data store API call to create the resource and once as Well Known Binary in the column which was created automatically for use in GeoServer. To convert a geometry to WKB, the Shapely library can be used.

The API call to upsert a record should be sent to <ckan url>/api/3/action/datastore_upsert

```
{
    "records": [
        {
            "situation_id": "just_testing_low",
            "version": "3",
            "severity": "low",
            "valid_from": "2021-11-01 00:00:00",
            "valid_to": "2021-12-31 23:59:59",
            "causes": "Gaten in de weg repareren. Lage prioriteit ",
            "lon": 5.2,
```

```
            "lat": 52.2,
            "the_geom_from_lon_lat": "01010000200400000000…"
        }
    ],
    "force": "true",
    "method": "upsert",
    "resource_id": "gebeurtenisinformatie"
}
```

## Test and deployment of the harvester

Once the harvester has been implemented, it should be tested by running it in standalone mode. After a successful test, the harvester can be deployed to a server where it can run reliably (so that is not a laptop which is constantly turned on and off). Schedule a task to periodically run the harvester. The schedule depends on a) the use case and b) the number of connections the 3rd party allows you to create before blacklisting you.