
invenio Documentation

Release 3.1.0.dev20181106

CERN

Dec 14, 2018

Contents

1	Quickstart	3
1.1	Launch an Invenio instance	3
1.2	Create, Display, Search Records	5
1.3	Next Steps	7
2	Build a data model	9
2.1	First steps	9
2.2	Define a JSONSchema	10
2.3	Define an Elasticsearch mapping	11
2.4	Naming JSONSchemas and mappings	12
2.5	Define a Marshmallow schema	14
2.6	Define serializers	14
2.7	Define loaders	15
2.8	Define templates	16
2.9	Configure the UI	16
2.10	Configure the REST API	17
2.11	Next steps	18
3	Tutorial: Develop a module	21
3.1	Scaffold new module	21
3.2	Install, run, test, document and publish	22
3.3	Form, views and templates	24
3.4	Publish on GitHub	29
3.5	Continue integration with TravisCI	30
4	Invenio module layout	33
4.1	*.rst files	33
4.2	setup.py	34
4.3	MANIFEST.in	34
4.4	run-tests.sh	34
4.5	docs folder	34
4.6	examples folder	34
4.7	tests folder	35
4.8	invenio_foo folder	35
5	Application architecture	37
5.1	Core concepts	37

5.2	Interfaces: WSGI, CLI and Celery	38
5.3	Application assembly	39
5.4	Assembly phases	39
5.5	Module discovery	40
5.6	WSGI: UI and REST	40
5.7	Implementation	40
6	Migrating to v3	41
6.1	Dumping data from v1.2	41
6.2	Loading data in v3	42
7	Login with ORCID	45
7.1	ORCID API credentials	45
7.2	Configuring Invenio	46
8	History	49
8.1	From software to framework	49
8.2	What happened to Invenio v2?	50
9	Releases	51
9.1	Maintenance Policy	51
9.2	Version 3.0.1	52
9.3	Version 3.0.0	53
9.4	Version 2.x	55
9.5	Version 1.x	56
9.6	Version 0.x	57
10	Community	59
10.1	Getting help	59
10.2	Communication channels	60
10.3	Contribution guide	61
10.4	Style guide	64
10.5	Developer environment guide	64
10.6	Translation guide	65
10.7	Maintainer's guide	66
10.8	Governance	68
10.9	Code of Conduct	71
10.10	License	72

Open Source framework for large-scale digital repositories.

1.1 Launch an Invenio instance

1.1.1 Prerequisites

To be able to develop and run Invenio you will need the following installed and configured on your system:

- [Docker](#) and [Docker Compose](#)
- [NodeJS v6.x+](#) and [NPM v4.x+](#)
- [Enough virtual memory](#) for Elasticsearch.

1.1.2 Overview

Creating your own Invenio instance requires scaffolding two code repositories using [Cookiecutter](#):

- one code repository for the main website.
- one code repository for the data model.

These code repositories will be where you customize and develop the features of your instance.

1.1.3 Bootstrap

Before we begin, you want to make sure to have Cookiecutter installed. Invenio leverages this tool to generate the starting boilerplate for different components, so it will be useful to have in general. We recommend you install it as a user package or in the virtualenv we define below.

```
# Install cookiecutter if it is not already installed
$ sudo apt-get install cookiecutter
# OR, once you have created a virtualenv per the steps below, install it
(my-repository-venv)$ pip install --upgrade cookiecutter
```

Note: If you install Cookiecutter in the virtualenv, you will need to activate the virtualenv to be able to use *cookiecutter* on the command-line.

We can now begin. First, let's create a *virtualenv* using *virtualenvwrapper* in order to sandbox our Python environment for development:

```
$ mkvirtualenv my-repository-venv
```

Now, let's scaffold the instance using the *official cookiecutter template*.

```
(my-repository-venv)$ cookiecutter gh:inveniosoftware/cookiecutter-invenio-instance --  
↪checkout v3.0  
# ...fill in the fields...
```

Now that we have our instance's source code ready we can proceed with the initial setup of the services and dependencies of the project:

```
# Fire up the database, Elasticsearch, Redis and RabbitMQ  
(my-repository-venv)$ cd my-site/  
(my-repository-venv)$ docker-compose up -d  
Creating network "mysite_default" with the default driver  
Creating mysite_cache_1 ... done  
Creating mysite_db_1 ... done  
Creating mysite_es_1 ... done  
Creating mysite_mq_1 ... done
```

If the Elasticsearch service fails to start mentioning that it requires more virtual memory, see the following fix <<https://www.elastic.co/guide/en/elasticsearch/reference/current/docker.html#docker-cli-run-prod-mode>>`_.

1.1.4 Customize

This instance doesn't have a data model defined, and thus it doesn't include any records you can search and display. To scaffold a data model for the instance we will use the *official data model cookiecutter template*:

```
(my-repository-venv)$ cd .. # switch back to the parent directory  
(my-repository-venv)$ cookiecutter gh:inveniosoftware/cookiecutter-invenio-datamodel -  
↪checkout v3.0  
# ...fill in the fields...
```

For the purposes of this guide, our data model folder is *my-datamodel*.

Let's also install the data model in our virtualenv:

```
(my-repository-venv)$ pip install -e .
```

Note: Once you publish your data model somewhere, i.e. the *Python Package Index*, you might want to edit your instance's *setup.py* file to add it there as a dependence.

Now that we have a data model installed we can create database tables and Elasticsearch indices:

```
(my-repository-venv)$ cd my-site  
(my-repository-venv)$ ./scripts/bootstrap  
(my-repository-venv)$ ./scripts/setup
```


1.1.5 Run

You can now run the necessary processes for the instance:

```
(my-repository-venv)$ ./scripts/server
* Environment: development
* Debug mode: on
* Running on https://127.0.0.1:5000/ (Press CTRL+C to quit)
```

You can now visit <https://127.0.0.1:5000/> !

1.2 Create, Display, Search Records

1.2.1 Create a record

By default, the data model has a records REST API endpoint configured, which allows performing CRUD and search operations over records. Let's create a simple record via `curl`, by sending a POST request to `/api/records` with some sample data:

```
$ curl -k --header "Content-Type: application/json" \
  --request POST \
  --data '{"title": "Some title", "contributors": [{"name": "Doe, John"}]}' \
  https://localhost:5000/api/records/?prettyprint=1
```

When the request was successful, the server returns the details of the created record:

```
{
  "created": "2018-05-23T13:28:19.426206+00:00",
  "id": 1,
  "links": {
    "self": "https://localhost:5000/api/records/1"
  },
  "metadata": {
    "contributors": [
      {
        "name": "Doe, John"
      }
    ],
    "id": 1,
    "title": "Some title"
  },
  "revision": 0,
  "updated": "2018-05-23T13:28:19.426213+00:00"
}
```

Note: Because we are using a self-signed SSL certificate to enable HTTPS, your web browser will probably display a warning when you access the website. You can usually get around this by following the browser's instructions in the warning message. For CLI tools like `curl`, you can ignore the SSL verification via the `-k/--insecure` option.

1.2.2 Display a record

You can now visit the record's page at <https://localhost:5000/records/1>, or fetch it via the REST API:

```
# You can find this URL under the "links.self" key of the previous response
$ curl -k --header "Content-Type: application/json" \
  https://localhost:5000/api/records/1?prettyprint=1

{
  "created": "2018-05-23T13:28:19.426206+00:00",
  "id": 1,
  "links": {
    "self": "https://localhost:5000/api/records/1"
  },
  "metadata": {
    "contributors": [
      {
        "name": "Doe, John"
      }
    ],
    "id": 1,
    "title": "Some title"
  },
  "revision": 0,
  "updated": "2018-05-23T13:28:19.426213+00:00"
}
```

1.2.3 Search for records

The record you created before, besides being inserted into the database, is also indexed in Elasticsearch and available for searching. You can search for it via the Search UI page at <https://localhost:5000/search>, or via the REST API from the `/api/records` endpoint:

```
$ curl -k --header "Content-Type: application/json" \
  https://localhost:5000/api/records/?prettyprint=1

{
  "aggregations": {
    "type": {
      "buckets": [],
      "doc_count_error_upper_bound": 0,
      "sum_other_doc_count": 0
    }
  },
  "hits": {
    "hits": [
      {
        "created": "2018-05-23T13:28:19.426206+00:00",
        "id": 1,
        "links": {
          "self": "https://localhost:5000/api/records/1"
        },
        "metadata": {
          "contributors": [
            {
              "name": "Doe, John"
            }
          ],
          "id": 1,

```

(continues on next page)

(continued from previous page)

```

        "title": "Some title"
    },
    "revision": 0,
    "updated": "2018-05-23T13:28:19.426213+00:00"
}
],
"total": 1
},
"links": {
    "self": "https://localhost:5000/api/records/?size=10&sort=mostrecent&page=1"
}
}

```

1.3 Next Steps

Although we can run and interact with the instance, we're not quite there yet in terms of having a proper Python package that's ready to be tested and deployed to a production environment.

You may have noticed that after running the `cookiecutter` command for the instance and the data model, there was a note for checking out some of the TODOs. You can run the following command in each code repository directory to see a summary of the TODOs again:

```
$ grep --color=always --recursive --context=3 --line-number TODO .
```

Let's have a look at some of them one-by-one and explain what they are for:

1. Creating a `requirements.txt`: This file is used for pinning the Python dependencies of your instance to specific versions in order to achieve reproducible builds when deploying your instance. You can generate this file in the following fashion (note, this is only for the *instance* and not the *data model*):

```

$ cd my-repository/
$ workon my-repository-venv
(my-repository-venv) $ pip install --editable .
(my-repository-venv) $ pip install pip-tools
(my-repository-venv) $ pip-compile

```

2. Python packages require a `MANIFEST.in` which specifies what files are part of the distributed package. You can update the existing file by running the following commands:

```

(my-repository-venv) $ git init
(my-repository-venv) $ git add --all
(my-repository-venv) $ pip install --editable .[all]
(my-repository-venv) $ check-manifest --update

```

3. Translations configuration (`.tx/config`): You might also want to generate the necessary files to allow localization of the instance in different languages via the [Transifex platform](#):

```

(my-repository-venv) $ python setup.py extract_messages
(my-repository-venv) $ python setup.py init_catalog -l en
(my-repository-venv) $ python setup.py compile_catalog

```

Ensure project has been created on Transifex under the `my-repository` organisation.

Install the `transifex-client`

```
(my-repository-venv) $ pip install transifex-client
```

Push source (.pot) and translations (.po) to Transifex:

```
(my-repository-venv) $ tx push --skip --translations
```

Pull translations for a single language from Transifex

```
# same error here
(my-repository-venv) $ tx pull --language en
```

1.3.1 Testing

In order to run tests for the instance, you can run:

```
# Install testing dependencies
$ workon my-repository-venv
# The following makes sure you have the tests dependencies installed
# if you already installed the instance via .[all] you can skip this install
(my-repository-venv) $ pip install --editable .[tests]
(my-repository-venv) $ ./run-tests.sh # will run all the tests...
# ...or to run individual tests
(my-repository-venv) $ pytest tests/ui/test_views.py
```

1.3.2 Documentation

In order to build and preview the instance's documentation, you can run the `setup.py build_sphinx` command:

```
$ workon my-repository-venv
# The following makes sure you have the docs dependencies installed
# if you already installed the instance via .[all] you can skip this install
(my-repository-venv) $ pip install --editable .[docs]
(my-repository-venv) $ python setup.py build_sphinx
```

Open up `docs/_build/html/index.html` in your browser to see the documentation.

CHAPTER 2

Build a data model

An Invenio data model, a bit simply put, defines *a record type*. You can also think of a data model as a supercharged database table.

In addition to storing records (aka documents or rows in a database) according to a specific structure, a data model also deals with:

- Access to records via REST APIs and landing pages.
- Internal storage, representation and retrieval of records and persistent identifiers.
- Mapping external representations to/from the internal representation via loaders and serializers.

You can build data models that are both custom to your exact needs, or you can build data models that follow standard metadata formats such as Dublin Core, DataCite or MARC21. In fact, a data model does not put any restrictions on what you can store, except that a record must be stored internally as JSON.

You can build data models for classic digital repository use cases such as bibliographic and author records, but Invenio is in no way limited to these classic use cases, and you could as well build your geographical research database on top of Invenio.

2.1 First steps

First of all, make sure you have followed the [Launch an Invenio instance](#), to ensure you have scaffolded an initial *Invenio instance* and *a data model package*.

You should see a directory structure similar to the one below in the newly scaffolded data model package:

```
|-- ...
|-- docs
|   |-- ...
|-- my_datamodel
|   |-- config.py
|   |-- jsonschemas/
|   |-- loaders/
```

(continues on next page)

(continued from previous page)

```
| |-- mappings/
| |-- marshmallow/
| |-- serializers/
| |-- static/
| |-- templates/
| `-- ...
|-- setup.py
`-- tests
    |-- ...
```

Building a data model involves the following tasks:

- Define a JSONSchema.
- Define an Elasticsearch mapping.
- Define a Marshmallow schema.
- Define serializers.
- Define loaders.
- Define templates.
- Choose a persistent identifier scheme.
- Configure the REST API and UI.

2.2 Define a JSONSchema

Internally records are stored as JSON, and in order to validate the structure of the stored JSON you must write a [JSONSchema](#).

The scaffolded data model package includes an example of a simple JSONSchema, that you can use to get a feeling of what a JSONSchema looks like.

```
|-- my_datamodel
|   |-- jsonschemas
|   |   |-- __init__.py
|   |   |-- records
|   |       |-- record-v1.0.0.json
```

In `record-v1.0.0.json` you should see something like:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "id": "https://localhost/schemas/records/record-v1.0.0.json",
  "type": "object",
  "properties": {
    "title": {
      "description": "Record title.",
      "type": "string"
    },
  },
}
```

Example record

An example record that validates against this schema could look like:

```
{
  "$schema": "https://localhost/schemas/records/record-v1.0.0.json",
  "title": "My record"
}
```

Note, that the `$schema` key points to the JSONSchema that the record should be validated against.

Discovery of schemas

Invenio is using standard Python entry points to discover your data model package's JSONSchemas. Thus, you'll see in the `setup.py` an entry point group `invenio_jsonschemas.schemas`:

```
setup(
    # ...
    entry_points={
        'invenio_jsonschemas.schemas': [
            'my_datamodel = my_datamodel.jsonschemas'
        ],
        # ...
    },
)
```

Note: A typical mistake is to forget to add a blank `__init__.py` file inside the `jsonschemas` folder, in which case the entry point won't work.

2.3 Define an Elasticsearch mapping

In order to make records searchable, the records need to be indexed in Elasticsearch. Similarly to the JSONSchema that allows you to validate the structure of the JSON, you need to define an *Elasticsearch mapping*, that tells Elasticsearch how to index your document.

The scaffolded data model package includes an example of a simple Elasticsearch mapping

```
|-- my_datamodel
|   |-- mappings
|   |   |-- __init__.py
|   |   |-- v5
|   |   |   |-- __init__.py
|   |   |   |-- records
|   |   |       |-- record-v1.0.0.json
|   |   |-- v6
|   |       |-- __init__.py
|   |       |-- records
|   |           |-- record-v1.0.0.json
```

Note, you need an Elasticsearch mapping per major version of Elasticsearch you want to support.

In `record-v1.0.0.json` you should see something like:

```
{
  "mappings": {
    "record-v1.0.0": {
      "date_detection": false,
      "numeric_detection": false,
```

(continues on next page)

(continued from previous page)

```
    "properties": {
      "$schema": {
        "type": "text",
        "index": false
      },
      "title": {
        "type": "text",
      },
      "keywords": {
        "type": "keyword"
      },
    },
  }
}
```

The above Elasticsearch mapping, similarly to the JSONSchema, defines the structure of the JSON, but also how it should be indexed.

For instance, in the above example the `title` field is of type `text`, which applies stemming when searching, whereas the `keywords` field is of type `keyword`, which means no stemming is applied. The mapping also allows you to define e.g. that a `lat` and a `lon` field are in fact geographical coordinates, and enable geospatial queries over your records.

2.4 Naming JSONSchemas and mappings

You may already have noticed that both JSONSchemas and Elasticsearch mappings are using the same folder structure and naming scheme:

```
|-- my_datamodel
|   |-- jsonschemas
|   |   |-- __init__.py
|   |   |-- records
|   |   |-- record-v1.0.0.json
|   |-- mappings
|   |   |-- __init__.py
|   |   |-- v6
|   |       |-- __init__.py
|   |       |-- records
|   |       |-- record-v1.0.0.json
```

The naming scheme is very important for three reasons:

1. Indexing of records
2. Data model evolution
3. Discovery of mappings

1. Indexing of records

Invenio will determine the Elasticsearch index for a given record, based on the record's `$schema` key. For instance, given the following record:

```
{
  "$schema": "https://localhost/schemas/records/record-v1.0.0.json",
```

(continues on next page)

(continued from previous page)

```

    "...": "..."
}

```

Invenio will send the above record to the `records-record-v1.0.0` Elasticsearch index. Note, it's possible to customize this behavior.

2. Data model evolution

Over time data models are likely to evolve. In many cases, you can simply make backward compatible changes to the existing JSONSchema and Elasticsearch mappings. In cases, where you change the data model in a backward incompatible way, you create a new JSONSchema and new mappings (e.g. `record-v1.1.0.json`)

```

|-- my_datamodel
|   |-- jsonschemas
|   |   |-- __init__.py
|   |   |-- records
|   |       |-- record-v1.0.0.json
|   |       |-- record-v1.1.0.json
|   |-- mappings
|   |   |-- __init__.py
|   |   |-- v6
|   |       |-- __init__.py
|   |       |-- records
|   |           |-- record-v1.0.0.json
|   |           |-- record-v1.1.0.json

```

This allows you to simultaneously store old and new records - i.e. you don't have to take down your service for hours to migrate millions of records from one version to a new one.

Now of course, old records will be sent to the `records-record-v1.0.0` index and new records will be sent to the `records-record-v1.1.0` index. However, a special Elasticsearch *index alias* `records` is also created, that allows you to search over both old and new records, thus smoothly handling data model evolution.

3. Discovery of mappings

Invenio is using standard Python entry points to discover your data model package's Elasticsearch mappings. Thus, you'll see in the `setup.py` an entry point group `invenio_search.mappings`:

```

setup(
    # ...
    entry_points={
        'invenio_search.mappings': [
            'records = my_datamodel.mappings'
        ],
        # ...
    },
)

```

Note, that the left-hand-side of the entry point, `records = my_datamodel.mappings`, defines the folder name/index alias (i.e. `records`) and that the right-hand-side defines the Python import path to the mappings package.

Note: A typical mistake is to forget to add a blank `__init__.py` file inside the mappings, v5 and v6 folders, in which case the entry points won't be correctly discovered.

2.5 Define a Marshmallow schema

[Marshmallow](#) is a Python library that helps you write highly advanced serialization/deserialization/validation rules for your input/output data. You can think of Marshmallow schemas as akin to form validation.

Marshmallow use in Invenio is optional, but is usually very helpful when you go beyond purely structural data validation - e.g. validating one field given the value of another field.

In Invenio, the Marshmallow schemas are located in the `marshmallow` Python module. You may have multiple Marshmallow schemas depending on your serialization and deserialization needs.

```
|-- my_datamodel
|   |-- marshmallow
|   |   |-- __init__.py
|   |   `-- json.py
```

Below is a simplified example of a Marshmallow schema you could use in `json.py` (note, the scaffolded data model package, includes a more complete example):

```
from invenio_records_rest.schemas import StrictKeysMixin
from marshmallow import fields

class RecordSchemaV1(StrictKeysMixin):
    metadata = fields.Raw()
    created = fields.Str()
    revision = fields.Integer()
    updated = fields.Str()
    links = fields.Dict()
    id = fields.Str()
```

In Invenio the Marshmallow schemas are often used together with serializers and loaders, so continue reading to see how the schema is used.

What's the difference: JSONSchemas, Mappings and Marshmallow?

It may seem a bit confusing that Invenio is dealing with three types of schemas. There's however good reasons:

- **JSONSchema:** Deals with the internal structural validation of records stored in the database (much like you define the table structure in database).
- **Elasticsearch mappings:** Deals with how records are indexed in Elasticsearch which has big impact on your search results ranking.
- **Marshmallow schema:** Deals with primarily data validation and transformation for both serialization and de-serialization (think of it as form validation).

2.6 Define serializers

Think of serializers as the definition of your output formats for records. The serializers are responsible for transforming the internal JSON for a record into some external representation (e.g. another JSON format or XML).

Serializers are defined in the `serializers` module:

```
|-- my_datamodel
|   |-- serializers
|   |   |-- __init__.py
```

By default, Invenio provides serializers that can help you serialize your internal record into common formats such as JSON-LD, Dublin Core, DataCite, MARCXML, Citation Style Language.

Example

In the scaffolded data model package, there's an example of a simple serializer:

```
from invenio_records_rest.serializers.json import \
    JSONSerializer
from invenio_records_rest.serializers.response import \
    record_responsify, search_responsify

from ..marshmallow import RecordSchemaV1

#: JSON serializer definition.
json_v1 = JSONSerializer(RecordSchemaV1, replace_refs=True)

#: Serializer for individual records.
json_v1_response = record_responsify(json_v1, 'application/json')
#: Serializer for search results.
json_v1_search = search_responsify(json_v1, 'application/json')
```

First, we create an instance of the `JSONSerializer` and provide it with our previously created `Marshmallow` schema. The `marshmallow` schema is used to transform the internal JSON prior to that the `JSONSerializer` dumps the actual JSON output. This allows you e.g. to evolve your internal data model, without affecting your REST API.

Next, we create two different **response serializers**: `json_v1_response` and `json_v1_search`. The former is responsible for producing an HTTP response for an individual record, while the latter is responsible for producing an HTTP response for a search result (i.e. multiple records).

The response serializer can not only output data to the HTTP response body, but can also add HTTP headers (e.g. Link headers).

You can see examples of the output from the two response serializers in the Quickstart section: [Display a record](#) and [Search for records](#).

2.7 Define loaders

Think of loaders as the definition of your input formats for records. You only need loaders if you plan to allow creation of records via the REST API.

The loaders are responsible for transforming a request payload (external representation) into the internal JSON format.

Loaders are defined in the `loaders` module:

```
|-- my_datamodel
|   |-- loaders
|   |-- __init__.py
```

Loaders are defined in much the same way as serializers, and similarly you can use the `Marshmallow` schemas:

```
from invenio_records_rest.loaders.marshmallow import \
    marshmallow_loader
from ..marshmallow import MetadataSchemaV1

json_v1 = marshmallow_loader(MetadataSchemaV1)
```

Note, that you are not required to use Marshmallow for deserialization, but it allows you to use advanced data validation rules on your REST API.

2.8 Define templates

In order to display records not only on your REST API, but also provide search interface and landing pages for your record you need to provide templates that render your records.

You will need two different types of templates:

- Search result template
- Landing page template

The templates are stored in two different folders (`static` and `templates`):

```
|-- my_datamodel
|   |-- static
|   |   |-- templates
|   |       |-- my_datamodel
|   |           |-- results.html
|   |-- templates
|   |   |-- my_datamodel
|   |       |-- record.html
```

Search result template

The Invenio search interface is run by a JavaScript application, and thus the template is rendered client side in the user's browser. The template uses data received by the REST API and thus your REST API must be able to deliver all information you would like to render in the template (your serializers are responsible for this).

The search results template is by default (it's configurable) located in `static/templates/my_datamodel/results.html` and is using the Angular template syntax.

Landing page template

The landing page for a single record is rendered on the server-side using a Jinja template.

The landing page template is by default (it's configurable) located in `templates/my_datamodel/record.html` and is using the Jinja template syntax.

2.9 Configure the UI

Last step after having defined all the different schemas, serializers, loaders and templates is to configure your REST API and landing pages for your records.

This is all done from the `config.py`:

```
|-- my_datamodel
|   |-- config.py
```

Landing page

Let's start by configuring the landing page:

```
RECORDS_UI_ENDPOINTS = {
    'recid': {
        'pid_type': 'recid',
        'route': '/records/<pid_value>',
        'template': 'my_datamodel/record.html',
    },
}
```

Here an explanation of the different keys:

- `pid_type`: Defines the persistent identifier type which the resolver should use to lookup records. Invenio provides an internal persistent identifier type called `recid` which is an auto-incrementing integer.
- `route`: URL endpoint under which to expose the landing pages.
- `template`: Template to use when rendering the landing page.
- `recid`: Unique name of the endpoint. If this is the primary landing page, it must be named the same as the value of `pid_type` (i.e. `recid`).

2.10 Configure the REST API

Configuring the REST API is done similarly to the landing pages via the `RECORDS_REST_ENDPOINTS` configuration variable in `config.py`:

Persistent identifier type

First you provide the persistent identifier type used by the resolver. You also need to configure a persistent identifier minter and fetcher. In the scaffolded data model package, you are just using the already provided `recid` minter and fetchers.

A `minter` is responsible for generating a new persistent identifier for your record, while a `fetcher` is responsible for extracting the persistent identifier from your search results:

```
RECORDS_REST_ENDPOINTS = {
    'recid': dict(
        pid_type='recid',
        pid_minter='recid',
        pid_fetcher='recid',
        # ...
    ),
}
```

Search

Next, you define the Elasticsearch index to use for searches. The index is defined as `records` because this is the index alias which was created for our mappings `records/record-v1.0.0.json` (see [Naming JSONSchemas and mappings](#)).

```
RECORDS_REST_ENDPOINTS = {
    'recid': dict(
        # ...
        search_index='records',
    ),
}
```

Serializers

Next, you define which serializers to use. Invenio is using HTTP Content Negotiation to choose your serializer. You have to specify the serializer for individual records in `record_serializers` and the serializers for search results in `search_serializers`:

```
RECORDS_REST_ENDPOINTS = {
    'recid': dict(
        # ...
        record_serializers={
            'application/json': (
                'my_datamodel.serializers:json_v1_response'),
        },
        search_serializers={
            'application/json': (
                'my_datamodel.serializers:json_v1_search'),
        },
    ),
}
```

Loaders

Next, you define the loaders to use. Similar to the serializers the loaders are selected based on HTTP Content Negotiation.

```
RECORDS_REST_ENDPOINTS = {
    'recid': dict(
        # ...
        record_loaders={
            'application/json': (
                'my_datamodel.loaders:json_v1'),
        },
    ),
}
```

URL routes

Last you define the URL routes under which to expose your records:

```
RECORDS_REST_ENDPOINTS = {
    'recid': dict(
        # ...
        list_route='/records/',
        item_route='/records/<pid(recid):pid_value>',
    ),
}
```

2.11 Next steps

Above is a quick walk through of the different steps to build a data model. In order to get more details on individual topics we suggest further reading:

- [Invenio-Records-REST](#)
- [Invenio-JSONSchemas](#)
- [Invenio-PIDStore](#)
- [Invenio-Records](#)

- JSONSchema
- Elasticsearch mappings
- Elasticsearch field types
- Marshmallow schemas

Tutorial: Develop a module

3.1 Scaffold new module

The easiest way to create a new Invenio module is to use our provided [Cookiecutter](#) template to scaffold the new module.

First, make sure you have Cookiecutter installed as per [Bootstrap](#).

Now we will create the files for the module. A module is basically a folder gathering all the files needed for its installation and execution. So, go where you want the directory to be created, and run the command:

```
$ cookiecutter gh:inveniosoftware/cookiecutter-invenio-module
```

This will first clone the template from git to your current directory. Then, Cookiecutter will ask you questions about the module you want to create:

```
project_name [Invenio-FunGenerator]: Invenio-Unicorn
project_shortname [invenio-unicorn]:
package_name [invenio_unicorn]:
github_repo [inveniosoftware/invenio-unicorn]:
description [Invenio module that adds more fun to the platform.]:
author_name [Nice Unicorn]:
author_email [info@inveniosoftware.org]: nice@unicorn.com
year [2017]:
copyright_holder [Nice Unicorn]:
copyright_by_intergovernmental [True]:
superproject [Invenio]:
transifex_project [invenio-unicorn]:
extension_class [InvenioUnicorn]:
config_prefix [UNICORN]:
```

The directory `invenio-unicorn` has been created containing the generated files. All modules follow the same layout which is described in the [Invenio module layout](#) section.

Once you have a grasp on the module layout, you can continue to the *Install, run, test, document and publish* section, to learn how to install your new module.

3.2 Install, run, test, document and publish

In this section, we are going to see how to install the module we just scaffolded, run the tests, run the example application and build the documentation.

Before that, we need to **stop** any running Invenio instance.

3.2.1 Install the module

First, create a virtualenv for the module:

```
$ mkvirtualenv my-module-venv
```

Installing the module is very easy, you just need to go to its root directory and *pip install* it:

```
(my-module-venv)$ cd invenio-unicorn/  
(my-module-venv)$ pip install --editable .[all]
```

Some explanations about the command:

- the `--editable` option is used for development. It means that if you change the files in the module, you won't have to reinstall it to see the changes. In a production environment, this option shouldn't be used.
- the `.` is in fact the path to your module. As we are in the root folder of the module, we can just say *here*, which is what the dot means.
- the `[all]` after the dot means we want to install all dependencies, which is common when developing. Depending on your use of the module, you can install only parts of it:
 - the default (nothing after the dot) installs the minimum to make the module run.
 - `[tests]` installs the requirements to test the module.
 - `[docs]` installs the requirements to build the documentation.
 - some modules have extra options.

You can chain them: `[tests, docs]`.

3.2.2 Run the tests

In order to run the tests, you need to have a valid git repository. The following step needs to be run only once. Go in the root folder of the module:

```
(my-module-venv)$ git init  
(my-module-venv)$ git add --all  
(my-module-venv)$ check-manifest --update
```

What we have done:

- change the folder into a git repository, so it can record the changes made to the files.
- add all the files to this repository.

- update the file `MANIFEST.in` (this file controls which files are included in your Python package when it is created and installed).

Now, we are able to run the tests:

```
(my-module-venv) $ ./run-tests.sh
```

Everything should pass as we didn't change any files yet.

3.2.3 Run the example application

The example application is a small app that presents the features of your module. The example application is useful during e.g. development to have a minimal application to test your module with. By default, it simply prints a welcome page. To try it, go into the `examples` folder and run:

```
(my-module-venv) $ ./app-setup.sh
(my-module-venv) $ ./app-fixtures.sh
(my-module-venv) $ export FLASK_APP=app.py FLASK_DEBUG=1
(my-module-venv) $ flask run
```

You can now open a browser and go to the URL <http://localhost:5000/> where you should be able to see a welcome page.

To clean the server, run the `./app-teardown.sh` script after killing the server.

3.2.4 Build the documentation

The documentation can be built with the `run-tests.sh` script, but you need to have installed *tests* requirements, and to run the tests. If you just want to build the documentation, you will only need the *docs* requirements (see the install section above). Make sure you are back at the root directory of the module and run:

```
(my-module-venv) $ python setup.py build_sphinx
```

Open `docs/_build/html/index.html` in browser and voilà, the documentation is there.

3.2.5 Publishing on GitHub

Before going further in the tutorial, we can publish your repository to GitHub. This allows to integrate e.g. TravisCI continuous integration system and have easy publishing of your module to PyPI afterwards.

First, create an empty repository in your GitHub account. Be sure to not generate any *.gitignore* or *README* files, as our code already has them. If you don't have a GitHub account, you can skip this step, it is only necessary if you plan to publish your module on PyPI.

Now, go into the root directory of your module, and run

```
(my-module-venv) $ git remote add origin URL-OF-YOUR-GITHUB-REPO
```

Now, we can commit and push the generated files:

```
(my-module-venv) $ git commit -am "Initial module structure"
(my-module-venv) $ git push --set-upstream origin master
```

Finally, we create a new branch to develop on it

```
(my-module-venv)$ git checkout -b dev
```

3.2.6 Next Step

In the above steps we have seen how to do the basic operations with our module. To add some more functionality to it, the following guide shows the steps to create a view that will create records given some data: *Form, views and templates*.

3.3 Form, views and templates

In this tutorial we'll see how to add data to our Invenio application. To accomplish this we will cover several parts of the development process such as:

- How to create a form
- How to create a new view
- How to add a utility function
- How to add new templates
- How to use Jinja2
- How to define your own JSON Schema

3.3.1 Flask extensions

It is important to understand that Invenio modules are just regular [Flask extensions](#). The Flask documentation contains extensive documentation on the APIs, design patterns and in general how to develop with Flask, and it is highly recommended that you follow Flask tutorials to understand the basics of Flask.

3.3.2 1. Create the form

First, let's create a Python module that contains the forms of our project, we will use [Flask-WTF](#).

In `invenio_unicorn/forms.py`

```
"""Forms module."""

from __future__ import absolute_import, print_function

from flask_wtf import FlaskForm
from wtforms import StringField, TextAreaField, validators


class RecordForm(FlaskForm):
    """Custom record form."""

    title = StringField(
        'Title', [validators.DataRequired()]
    )
    description = TextAreaField(
```

(continues on next page)

(continued from previous page)

```

        'Description', [validators.DataRequired()]
    )

```

3.3.3 2. Create the views

In `invenio_unicorn/views.py` we'll create the endpoints for

- create: Form template
- success: Success template

and register all the views to our application.

```

"""Invenio module that adds more fun to the platform."""

from __future__ import absolute_import, print_function

from flask import Blueprint, redirect, render_template, request, url_for
from flask_babellex import gettext as _
from invenio_records import Record
from invenio_records.models import RecordMetadata

from .forms import RecordForm
from .utils import create_record

blueprint = Blueprint(
    'invenio_unicorn',
    __name__,
    template_folder='templates',
    static_folder='static',
)

@blueprint.route("/")
def index():
    """Basic view."""
    return render_template(
        "invenio_unicorn/index.html",
        module_name=__('Invenio-Unicorn'))

@blueprint.route('/create', methods=['GET', 'POST'])
def create():
    """The create view."""
    form = RecordForm()
    # if the form is valid
    if form.validate_on_submit():
        # create the record
        create_record(
            dict(
                title=form.title.data,
                description=form.description.data
            )
        )
        # redirect to the success page
        return redirect(url_for('invenio_unicorn.success'))

```

(continues on next page)

(continued from previous page)

```

records = _get_all()
return render_template('invenio_unicorn/create.html', form=form, records=records)

def _get_all():
    """Return all records."""
    return [Record(obj.json, model=obj) for obj in RecordMetadata.query.all()]

@blueprint.route("/success")
def success():
    """The success view."""
    return render_template('invenio_unicorn/success.html')

```

3.3.4 3. Create the templates

And now, let's create the templates.

We create a *create.html* template in `invenio_unicorn/templates/invenio_unicorn/` where we can override the `page_body` block, to place our form:

```

{% extends config.UNICORN_BASE_TEMPLATE %}

{% macro errors(field) %}
    {% if field.errors %}
    <span class="help-block">
        <ul class="errors">
            {% for error in field.errors %}
            <li>{{ error }}</li>
            {% endfor %}
        </ul>
    </span>
    {% endif %}
{% endmacro %}

{% block page_body %}
    <div class="container">
        <div class="row">
            <div class="col-md-12">
                <div class="alert alert-warning">
                    <b>Heads up!</b> This example is for demo proposes only
                </div>
                <h2>Create record</h2>
            </div>
            <div class="col-md-offset-3 col-md-6 well">
                <form action="{{ url_for('invenio_unicorn.create') }}" method="POST">
                    <div class="form-group {{ 'has-error' if form.title.errors }}">
                        <label for="title">{{ form.title.label }}</label>
                        {{ form.title(class_="form-control")|safe }}
                        {{ errors(form.title) }}
                    </div>
                    <div class="form-group {{ 'has-error' if form.description.errors }}">
                        <label for="description">{{ form.description.label }}</label>
                        {{ form.description(class_="form-control")|safe }}
                    </div>
                </form>
            </div>
        </div>
    </div>
{% endblock %}

```

(continues on next page)

(continued from previous page)

```

        {{ errors(form.description) }}
    </div>
    {{ form.csrf_token }}
    <button type="submit" class="btn btn-default">Submit</button>
</form>
</div>
</div>
<hr />
<div class="row">
    <div class="col-md-12">
        {% if records %}
        <h2>Records created</h2>
        <ol id="custom-records">
            {% for record in records %}
            <li>{{ record.title }}</li>
            {% endfor %}
        </ol>
        {% endif %}
    </div>
</div>
</div>
{% endblock page_body %}

```

And finally, the *success.html* page in *invenio_unicorn/templates/invenio_unicorn/* which will be rendered after a record is created.

```

{% extends config.UNICORN_BASE_TEMPLATE %}

{% block page_body %}
    <div class="container">
        <div class="row">
            <div class="col-md-12">
                <div class="alert alert-success">
                    <b>Success!</b>
                </div>
                <a href="{{ url_for('invenio_unicorn.create') }}" class="btn btn-warning">
→ Create more</a>
                <hr />
                <center>
                    <iframe src="//giphy.com/embed/WZmgVLMt7mp44" width="480" height="480"
→ frameBorder="0" class="giphy-embed" allowFullScreen></iframe><p><a href="http://
→ giphy.com/gifs/kawaii-colorful-unicorn-WZmgVLMt7mp44">via GIPHY</a></p>
                    </center>
                </div>
            </div>
        </div>
    </div>
{% endblock page_body %}

```

3.3.5 4. Create the record creation function

The *utils.py* file contains all helper functions of our module, so let's write the first utility that will create a record.

In *invenio_unicorn/utils.py*

```
"""Utils module."""
from __future__ import absolute_import, print_function

import uuid

from flask import current_app

from invenio_db import db
from invenio_indexer.api import RecordIndexer
from invenio_pidstore import current_pidstore
from invenio_records.api import Record

def create_record(data):
    """Create a record.

    :param dict data: The record data.
    """
    indexer = RecordIndexer()
    # create uuid
    rec_uuid = uuid.uuid4()
    # add the schema
    data["$schema"] = \
        current_app.extensions['invenio-jsonschemas'].path_to_url(
            'records/custom-record-v1.0.0.json'
        )
    # create PID
    current_pidstore.minters['recid'](rec_uuid, data)
    # create record
    created_record = Record.create(data, id_=rec_uuid)
    db.session.commit()

    # index the record
    indexer.index(created_record)
```

3.3.6 5. Create the custom-record JSON Schema

As you can see, our records use a custom schema. To define and use this schema, we need to write the `custom-record-v1.0.0.json` inside the `records` folder of your data model project (`my-datamodel` from the Quickstart tutorial [Customize](#)).

In `my-datamodel/my-datamodel/jsonschemas/records/custom-record-v1.0.0.json`

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "id": "http://localhost/schemas/records/custom-record-v1.0.0.json",
  "additionalProperties": true,
  "title": "my-datamodel v1.0.0",
  "type": "object",
  "properties": {
    "title": {
      "description": "Record title.",
      "type": "string"
    },
    "description": {
      "description": "Record description.",
```

(continues on next page)

(continued from previous page)

```

    "type": "string"
  },
  "id": {
    "description": "Invenio record identifier (integer).",
    "type": "string"
  }
},
"required": [
  "title",
  "description"
]
}

```

Demo time

Let's now see our Invenio module in action when integrated with our Invenio instance.

First we install our new Invenio-Unicorn module. For the purposes of this guide, our instance folder is *my-site*, and it's placed in the same root folder as *invenio-unicorn*.

```

$ workon my-repository-venv
(my-repository-venv)$ pip install --editable .[all]

```

Then, if you've followed the steps in the Quickstart guide, you can go to the instance folder, *my-repository*, and start the server script:

```
(my-repository-venv)$ cd ../my-site (my-repository-venv)$ ./scripts/server
```

Then go to `http://localhost:5000/create` and you will see the form we just created. There are two fields Title and Description.

Let's try the form, add something to the Title and click submit, you will see the validation errors on the form, fill in the Description and click submit. The form is now valid and it navigates you to the `/success` page.

3.4 Publish on GitHub

Now that we finished the development, we want to push the changes to GitHub.

3.4.1 Commit the code

To do so, we will first list our changes and add them to our local git repository:

```

(my-module-venv)$ git status
# shows all the files that have been modified
(my-module-venv)$ git add .
# adds all the modifications

```

Let's test our changes before we publish them. See [Run the tests](#) for more information.

```
(my-module-venv)$ ./run-tests.sh
```

If it complains about the manifest, it is because we added new files, but we didn't register them into the `MANIFEST.in` file, so let's do so:

```
(my-module-venv) $ check-manifest -u
```

3.4.2 Push the code

Once all the tests are passing, we can push our code. As we were developing on a branch created locally, we need to push the branch on GitHub:

```
(my-module-venv) $ git commit -am "New form, views and templates"
(my-module-venv) $ git push --set-upstream origin dev
```

3.5 Continue integration with TravisCI

This section will show you how to enable continue integration using the TravisCI service. Note, that you code must have been published on GitHub in order for this to work.


TravisCI allows to run the tests of your module in a test matrix with different Python versions, and different versions of dependent Python packages. This ensures that your module becomes more robust.

3.5.1 Configure travis-ci.org

1. Create an account

We need first to create an account on travis-ci.org.

Travis CI Blog Status Help

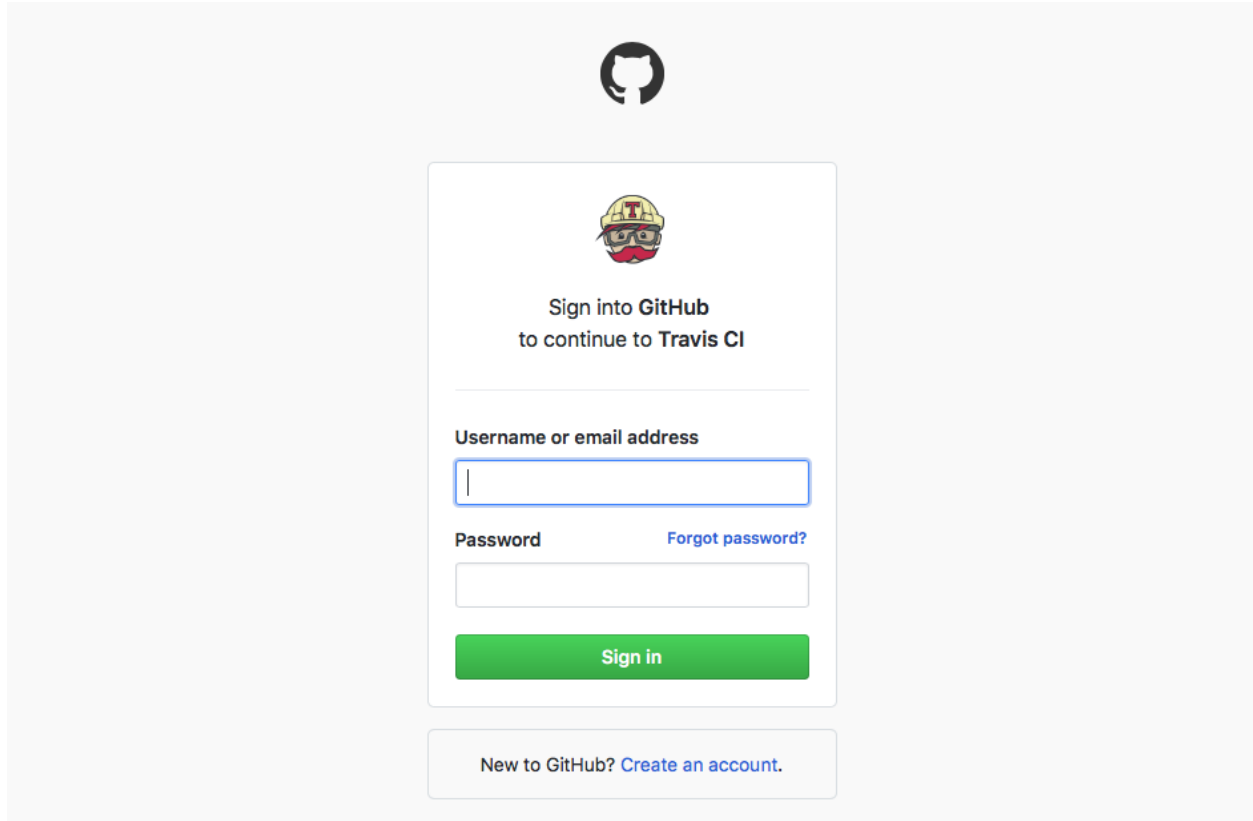
Sign in with GitHub 

Test and Deploy with Confidence

Easily sync your GitHub projects with Travis CI and you'll be testing your code in minutes!

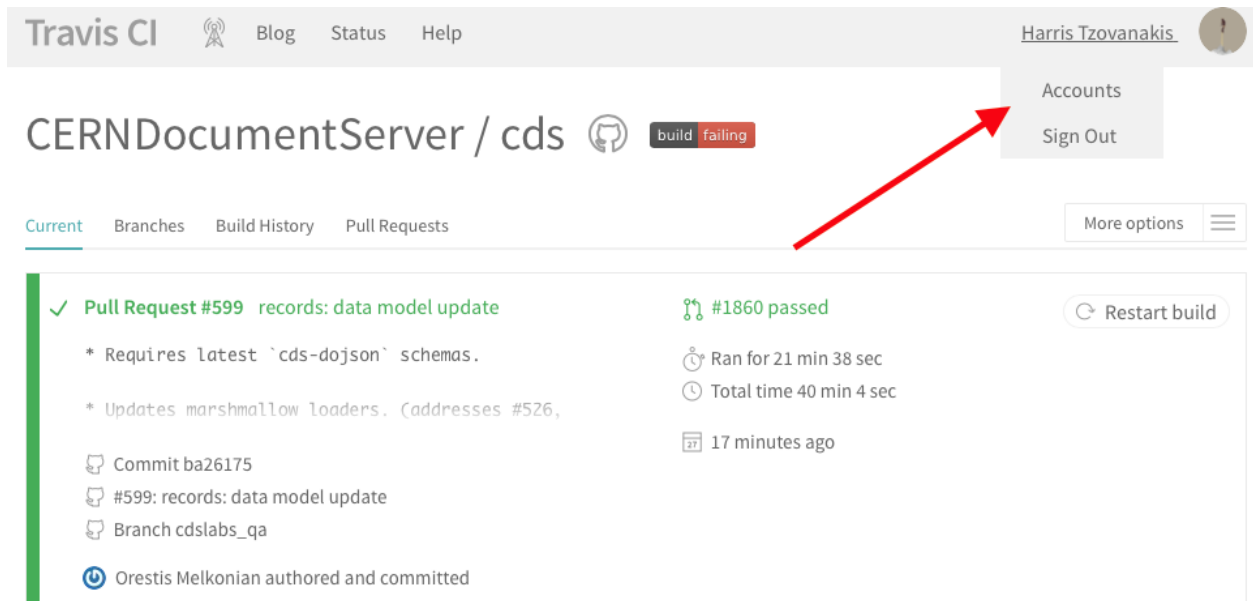


Add your github credentials to signup



2. Enable travis for your repo

Go to your account



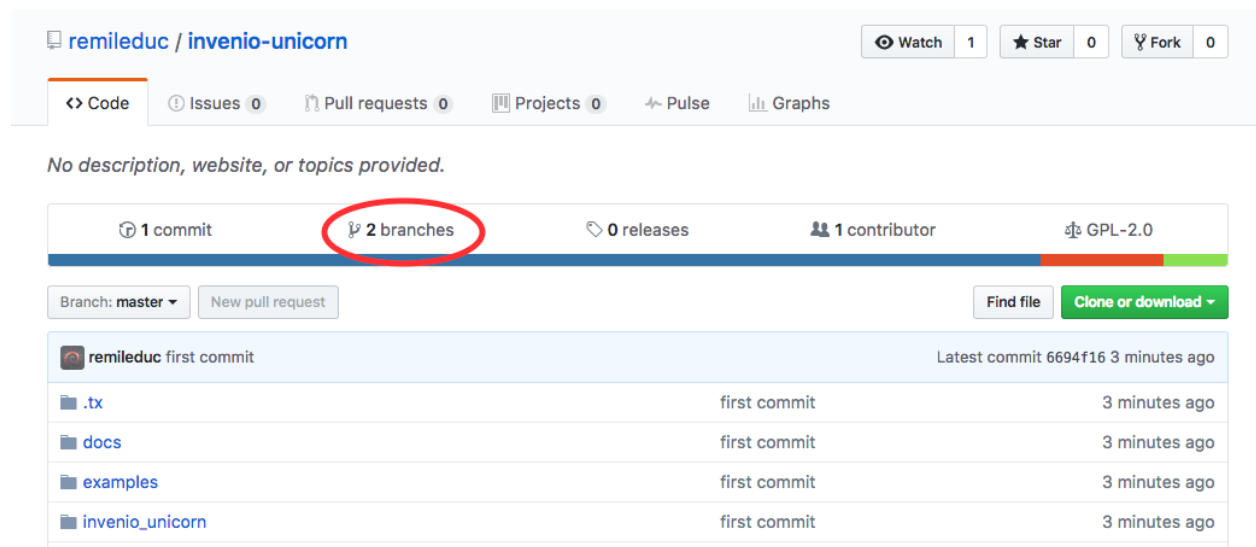
Click to enable the repo you have created



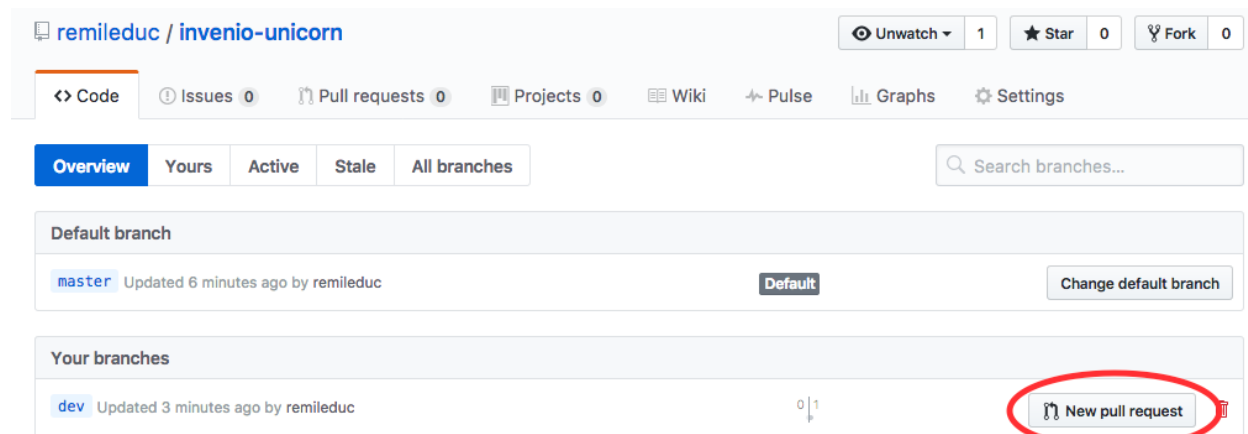
Done!

3.5.2 Create a Pull Request (PR)

We want that our changes get merged into the main branch (master) of the repository. So, let's go to the GitHub repository. From here, you can click on the *branch* button.



Then, click on *New pull request*



Now, you can check the differences that you will add to the main branch. Fill a description and create the pull request.

If TravisCI was correctly enabled, the pull request will now display a status check, that TravisCI is running the tests. If the tests pass, the status check will become green, if the tests fails it will become red. This way you're always sure not to integrated code that breaks your tests.

Invenio module layout

This page summarizes the standard structure and naming conventions of a module in Invenio v3.0. It serves as a reference point when developing a new module or enhancing an existing one.

A simple module may have the following folder structure:

```
invenio-foo/  
  docs/  
  examples/  
  invenio_foo/  
    templates/invenio_foo/  
    __init__.py  
    config.py  
    ext.py  
    version.py  
    views.py  
  tests/  
  *.rst  
  run-tests.sh  
  setup.py
```

These files are described in the sections below.

4.1 *.rst files

All these files are used by people who want to know more about your module (mainly developers).

- `README.rst` is used to describe your module. You can see the short description written in the Cookiecutter [here](#). You should update it with more details.
- `AUTHORS.rst` should list all contributors to this module.
- `CHANGES.rst` should be updated at every release and store the list of versions with the list of changes (changelog).

- `CONTRIBUTING.rst` presents the rules to contribute to your module.
- `INSTALL.rst` describes how to install your module.

4.2 setup.py

First, there is the `setup.py` file, one of the most important: this file is executed when you install your module with *pip*. If you open it, you can see different parts.

On the top, the list of the requirements:

- for normal use
- for development
- for tests

Depending on your needs, you can install only part of the requirements, or everything (`pip install invenio-foo[all]`).

Then, in the `setup()` function, you have the description of your module with the values entered in the Cookiecutter. At the end, you can find the `entrypoints` section. For the moment, there is only the registration in the Invenio application, and the translations.

4.3 MANIFEST.in

This file lists all the files included in the sub-folders. This file should be updated before the first commit. See the *Install, run, test, document and publish* section.

4.4 run-tests.sh

This is used to run a list of tests locally, to make sure that your module works as intended. It will generate the documentation, run *pytest* and do other checks.

4.5 docs folder

This folder contains the settings to generate documentation for your module, along with files where you can write the documentation. When you run the `run-tests.sh` script, it will create the documentation in HTML files in a sub-folder.

4.6 examples folder

Here you can find a small example of how to use your module. You can test it, follow the steps described in the *Run the example application* section

4.7 tests folder

Here are all the tests for your application, that will be run when you execute the `run-tests.sh` script. If all these tests pass, you can safely commit your work.

See [pytest-invenio](#) for how to structure your tests.

4.8 invenio_foo folder

This folder has the name of your module, in lower case with the dash changed into an underscore. Here is the code of your module. You can add any code files here, organized as you wish.

The files that already exist are kind of a standard, we are going through them in the following sections. A rule of thumb is that if you need multiple files for one action (for instance, 2 `views`: one for the API and a standard one), create a folder having the name of the file you want to split (here, a `views` folder with `ui.py` and `api.py` inside).

4.8.1 config.py

All configuration variables should be declared in this file.

4.8.2 ext.py

This file contains a class that extends the Invenio application with your module. It registers the module during the initialization of the application and loads the default configuration from `config.py`.

4.8.3 version.py

File containing the version of your module.

4.8.4 views.py

Here you declare the views or end points you want to expose. By default, it creates a simple view on the root end point that renders a template.

4.8.5 templates

All your Jinja templates should be stored in this folder. A Jinja template is an HTML file that can be modified according to some parameters.

4.8.6 static

If your module contains JavaScript or CSS files, they should go in a folder called `static`. Also, if you want to group them in bundles, you should add a `bundles.py` file next to the `static` folder.

Module naming conventions

Invenio modules are standalone independent components that implement some functionality used by the rest of the Invenio ecosystem. The modules provide API to other modules and use API of other modules.

A module is usually called:

1. with plural noun, meaning “database (of things)”, for example `invenio-records`, `invenio-tags`, `invenio-annotations`,
2. with singular noun, meaning “worker (using things)”, for example `invenio-checker`, `invenio-editor`.

A module may have split its user interface and REST API interface, for example `invenio-records-ui` and `invenio-records-rest`, to clarify dependencies and offer easy customisation.

Application architecture

Invenio is at the core an application built on-top of the Flask web development framework, and fully understanding Invenio's architectural design requires you to understand core concepts from Flask which will briefly be covered here.

The Flask application is exposed via different *application interfaces* depending on if the application is running in a webserver, CLI or job queue.

Invenio adds a powerful *application factory* on top of Flask, which takes care of dynamically assembling an Invenio application from the many individual modules that makes up Invenio, and which also allow you to easily extend Invenio with your own modules.

5.1 Core concepts

We will explain the core Flask concepts using simple Flask application:

```
from flask import Blueprint, Flask, request

# Blueprint
bp = Blueprint('bp', __name__)

@bp.route('/')
def my_user_agent():
    # Executing inside request context
    return request.headers['User-Agent']

# Extension
class MyExtension(object):
    def __init__(self, app=None):
        if app:
            self.init_app(app)

    def init_app(self, app):
        app.config.setdefault('MYCONF', True)
```

(continues on next page)

(continued from previous page)

```
# Application
app = Flask(__name__)
ext = MyExtension(app)
app.register_blueprint(bp)
```

You can save above code in a file `app.py` and run the application:

```
$ pip install Flask
$ export FLASK_APP=app.py flask run
```

Application and blueprint

Invenio is a large application built up of many smaller individual modules. The way Flask allows you to build modular applications is via *blueprints*. In above example we have a small blueprint which just have one *view* (`my_user_agent`), which returns the browser's user agent sting.

This blueprint is *registered* on the *Flask application*. This allow you to possible reuse the blueprint in another Flask application.

Flask extensions

Like blueprints allow you to modularise your Flask application's views, then Flask extensions allow you to modularise non-view specific initialization of your application (e.g. providing database connectivity).

Flask extensions are just objects like the one in the example below, which has `init_app` method.

Application and request context

Code in a Flask application can be executed in two “states”:

- *Application context*: when the application is e.g. being used via a CLI or running in a job queue (i.e. not handling requests).
- *Request context*: when the application is handling a request from a user.

In above example e.g. the code inside the view `my_user_agent` is executed during a request, and thus you can have access to the browser's user agent string. On the other hand, if you tried to access `request.headers` outside the view, the application would fail as no request is being processed.

The `request` object is a proxy object which points to the current request being processed. There is some magic happening behind the scenes in order to make this thread safe.

5.2 Interfaces: WSGI, CLI and Celery

Overall the Flask application is running via three different applications interfaces:

- **WSGI**: The frontend webservers interfaces with Flask via Flask's WSGI application.
- **CLI**: The command-line interface is made using Click and takes care of executing commands inside the Flask application.

- **Celery:** The distributed job queue is made using Celery and takes care of executing jobs inside the Flask application.

5.3 Application assembly

In each of the above interfaces, a Flask application needs to be created. A common pattern for large Flask applications is to move the application creation into a factory function, named an **application factory**.

Invenio provides a powerful application factory for Flask which is capable of dynamically assembling an application. In order to illustrate the basics of what the Invenio application factory does, have a look at the following example:

```
from flask import Flask, Blueprint

# Module 1
bp1 = Blueprint(__name__, 'bp1')
@bp1.route('/')
def hello():
    return 'Hello'

# Module 2
bp2 = Blueprint(__name__, 'bp1')
@bp2.route('/')
def world():
    return 'World'

# Application factory
def create_app():
    app = Flask(__name__)
    app.register_blueprint(bp1)
    app.register_blueprint(bp2)
    return app
```

The example illustrates two blueprints, which are statically registered on the Flask application blueprint inside the application factory. It is essentially this part that the Invenio application factory takes care of for you. Invenio will automatically discover all your installed Invenio modules and register them on your application.

5.4 Assembly phases

The Invenio application factory assembles your application in five phases:

1. **Application creation:** Besides creating the Flask application object, this phase will also ensure your instance folder exists, as well as route Python warnings through the Flask application logger.
2. **Configuration loading:** In this phase your application will load your instance configuration. Your instance configuration is essentially all the configuration variables where you don't want to use the default values, e.g. the database host configuration.
3. **URL converter loading:** In this phase, the application will load any of your URL converts. This phase is usually only needed for some few specific cases.
4. **Flask extensions loading:** In this phase all the Invenio modules which provides Flask extensions will initialize the extension. Usually the extensions will provide default configuration values they need, unless the user already set them.
5. **Blueprints loading:** After all extensions have been loaded, the factory will end with registering all the blueprints provided by the Invenio modules on the application.

Understanding above application assembly phases, what they do, and how you can plug into them is essential for fully mastering Invenio development.

Note: No loading order within a phase

It's very important to know, that within each phase, there is **no order** in how the Invenio modules are loaded. Say, with in the Flask extensions loading phase, there's no way to specify that one extension has to be loaded before another extension.

You only have the order of the phases to work, so e.g. Flask extensions are loaded before any blueprints are loaded.

5.5 Module discovery

In each of the application assembly phases, the Invenio factory automatically discover your installed Invenio modules. The way this works, is via Python **entry points**. When you install the Python package for an Invenio module, the package describes via entry points which Flask extensions, blueprints etc. that this module provides.

5.6 WSGI: UI and REST

Each of the application interfaces (WSGI, CLI, Celery) may need slightly different Flask applications. The Invenio application factory is in charge of assembling these applications, which is done through the five assembly phases.

The WSGI application is however also split up into two Flask applications:

- **UI:** Flask application responsible for processing all user facing views.
- **REST:** Flask application responsible for processing all REST API requests.

The reason to split the frontend part of Invenio into two separate applications is partly

- to be able to run the REST API in one domain (`api.example.org`) and the UI app on another domain (`www.example.org`)
- because UI and REST API applications usually have vastly different requirements.

As an example, a 404 Not found HTTP error, usually needs to render a template in the UI application, but return a JSON response in the REST API application.

5.7 Implementation

The following Invenio modules are each responsible for implementing parts of above application architecture, and it is highly advisable to dig deeper into these modules if you want a better understanding of the Invenio application architecture:

- **Invenio-Base:** Implements the Invenio application factory.
- **Invenio-Config:** Implements the configuration loading phase.
- **Invenio-App:** Implements default applications for WSGI, CLI and Celery.

Warning: Invenio v3 is significantly different from v1 and thus migrating from v1 to v3 is a complex operation. This guide will help you dump records and files from your v1 installation. You will need to write code to import the dumped data into your v3 installation. This is necessary because v3 support many different data models and thus you need to map your v1 MARC21 records into your new data model in v3.

6.1 Dumping data from v1.2

The module [Invenio-Migrator](#) will help you dump your v1 data and as well import the data in v3.

6.1.1 Install Invenio-Migrator in v1

There are several ways of installing Invenio-Migrator in your Invenio v1.2 or v2.1 production environment, the one we recommend is using [Virtualenv](#) to avoid any interference with the currently installed libraries:

```
$ sudo pip install virtualenv virtualenvwrapper
$ source /usr/local/bin/virtualenvwrapper.sh
$ mkvirtualenv migration --system-site-packages
$ workon migration
$ pip install invenio-migrator --pre
$ inveniomigrator dump --help
```

It is important to use the option `--system-site-packages` as Invenio-Migrator will use Invenio legacy python APIs to perform the dump. The package `virtualenvwrapper` is not required but it is quite convenient.

6.1.2 Dump records and files

```
$ mkdir /vagrant/dump
$ cd /vagrant/dump/
$ inveniomigrator dump records
```

This will generate one or more JSON files containing 1000 records each, with the following information:

- The record id.
- The record metadata, stored in the `record` key there is a list with one item for each of the revisions of the record, and each item of the list contains the MARC21 representation of the record plus the optional JSON.
- The files linked with the record, like for the record metadata it is a list with all the revisions of the files.
- Optionally it also contains the collections the record belongs to.

For more information about how to dump records and files see the [Usage section](#) of the Invenio-Migrator documentation.

The file path inside the Invenio legacy installation will be included in the dump and used as file location for the new Invenio v3 installation. If you are able to mount the file system following the same pattern in your Invenio v3 machines, there shouldn't be any problem, but if you can't do it, then you need to copy over the files folder manually using your favorite method, i.e.:

```
$ cd /opt/invenio/var/data
$ tar -zcvf /vagrant/dump/files.tar.gz files
```

Pro-tip: Maybe you want to have different data models in your new installation depending on the nature of the record, i.e. bibliographic records vs authority records. In this case one option is to dump them in different files using the `--query` argument when dumping from your legacy installation:

```
$ inveniomigrator dump records --query '-980__a:AUTHORITY' --file-prefix bib
$ inveniomigrator dump records --query '980__a:AUTHORITY' --file-prefix auth
```

6.1.3 Things

The dump command of the Invenio-Migrator works with, what we called, *things*. A *thing* is an entity you want to dump from your Invenio legacy installation, e.g. in the previous example the *thing* was *records*.

The list of *things* Invenio-Migrator can dump by default is listed via entry-points in the `setup.py`, this not only help us add new dump scripts easily, but also allows anyone to create their own dumpscripts from outside the Invenio-Migrator.

You can read more about which *things* are already supported by the [Invenio-Migrator documentation](#).

6.2 Loading data in v3

6.2.1 Install Invenio-Migrator in v3

Invenio-Migrator can be installed in any Invenio v3 environment using PyPI and the extra dependencies `loader`:

```
$ pip install invenio-migrator[loader]
```

Depending on what you want to load you might need to have installed other packages, i.e. to load communities from Invenio v2.1 you need `invenio-communities` installed.

This will add to your Invenio application a new set of commands under `dumps`:

```
$ invenio dumps --help
```

6.2.2 Load records and files

```
$ invenio dumps loadrecords /vagrant/dump/records_dump_0.json
```

This will generate one celery task to import each of the records inside the dump.

Pro-tip: By default Invenio-Migrator uses the bibliographic MARC21 standard to transform and load the records, we now that this might not be the case to all Invenio v3 installation, i.e authority records. By changing `MIGRATOR_RECORDS_DUMP_CLS` and `MIGRATOR_RECORDS_DUMPLOADER_CLS` you can customize the behavior of the loading command. There is a full chapter in the Invenio-Migrator documentation about [customizing loading](#) if you want more information.

6.2.3 Loaders

Each of the entities that can be loaded by Invenio-Migrator have a companion command generally prefixed by *load*, e.g. `loadrecords`.

The loaders are similar to the things we describe previously, but in this case, instead of entry-points, if you want to extend the default list of loaders it can be done adding a new command to `dumps`, more information about the loaders can be found in the [Invenio-Migrator documentation](#) and on how to add more commands in the [click documentation](#).

CHAPTER 7

Login with ORCID

ORCID provides a persistent identifiers for researchers and through integration in key research workflows such as manuscript and grant submission, supports automated linkages between you and your professional activities ensuring that your work is recognized.

This guide will show you how to enable your users to login with their ORCID account in Invenio. The underlying authentication protocol is based on OAuth which is the same used for enabling other social logins (like login with Twitter, Google, etc).

7.1 ORCID API credentials

In order to integrate your Invenio instance with ORCID the first step is to [apply for a client id/secret](#) to access the ORCID Sandbox.

Please, follow the official ORCID documentation for this part.

After successful application for an API client application key you should receive in your inbox an email with your `Client ID` and `Client secret`. You will need this information when configuring Invenio in the next step.

7.1.1 Redirect URI

After a user have authenticated on the ORCID site, the user will be redirected to a given page on the Invenio side. ORCID requires you to provide a list of authorized URI prefixes that could be allowed for this redirection to happen.

Depending on the `SERVER_NAME` used to configure the Invenio installation, you should fill this parameter with:

```
https://<SERVER_NAME>/oauth/authorized/orcid/
```

7.2 Configuring Invenio

In order to enable OAuth authentication for ORCID, just add these line to your `var/instance/invenio.cfg` file.

```
from invenio_oauthclient.contrib import orcid

OAUTHCLIENT_REMOTE_APPS = dict(
    orcid=orcid.REMOTE_SANDBOX_APP,
)

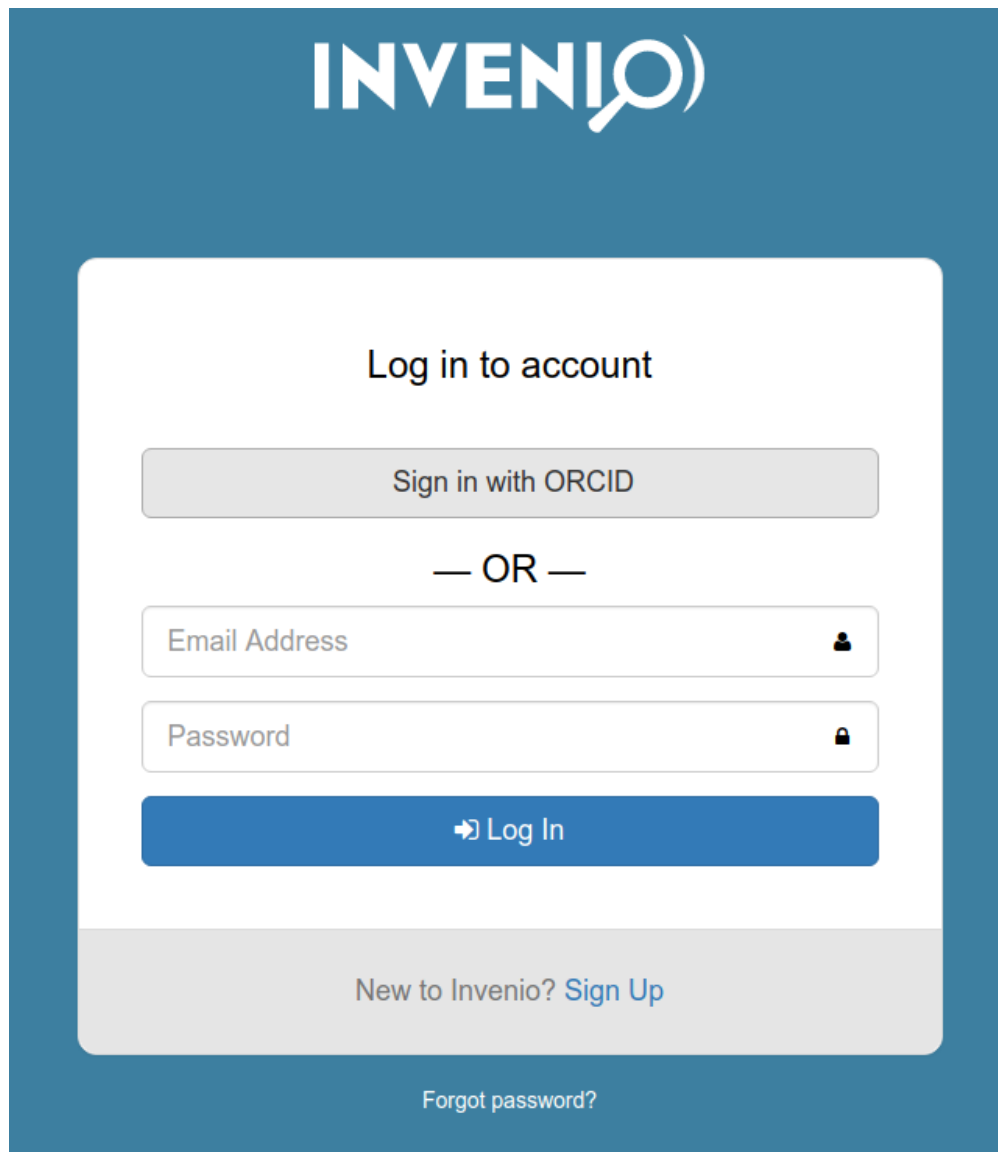
ORCID_APP_CREDENTIALS = dict(
    consumer_key="<your-orcid-client-id>",
    consumer_secret="<your-orcid-client-secret>",
)
```

where the client id and client secret are those provided by ORCID in the previous step.

If you can now visit:

```
https://<SERVER_NAME>/login
```

you will be able to see ORCID authentication enabled:



The image shows the Invenio login page. At the top, the Invenio logo is displayed in white on a blue background. Below the logo, the text "Log in to account" is centered. There are two main login options: "Sign in with ORCID" and a standard email/password login. The "Sign in with ORCID" option is a light gray button. Below it, the text "— OR —" is centered. The standard login form consists of two input fields: "Email Address" and "Password". The "Email Address" field has a user icon on the right, and the "Password" field has a lock icon on the right. Below these fields is a blue "Log In" button with a right arrow icon. At the bottom of the login form, there is a light gray bar with the text "New to Invenio? Sign Up". Below this bar, the text "Forgot password?" is centered.

INVENIO

Log in to account

Sign in with ORCID

— OR —

Email Address

Password

Log In

New to Invenio? [Sign Up](#)

[Forgot password?](#)

7.2.1 Sign-up on first login

The first time a user try to login with ORCID, they will be required to provide a username and email address. This is because ORCID does provide this information and it is required by Invenio in order to register an account.

8.1 From software to framework

Invenio v3 is a completely new framework that has been rewritten from scratch. Why such a dramatic decision? To understand why the rewrite was necessary we have to go back to when Invenio was called CDSWare, back to August 1st 2002 when the first version of Invenio was released.

In 2002:

- First iPod had just hit the market (Nov 2001).
- The Budapest Open Access Initiative had just been signed (Feb, 2002).
- JSON had just been discovered (2001).
- Python 2.1 had just been released (2001).
- Apache Lucene had just joined the Apache Jakarta project (but not yet an official top-level project).
- MySQL v4.0 beta was released and did not even have transactions yet.
- Hibernate ORM was released.
- The first DOI had just been assigned to a dataset.

Following products did not even exists:

- Apache Solr (2004)
- Google Maps (2005)
- Google Scholar (2004)
- Facebook (2007)
- Django (2005)

A lot has happen since 2002. Many problems that Invenio originally had to deal with now have open source off-the-shelf solutions available. In particular two things happen:

- Search become pervasive with the exponential growth of data collected and created on the internet every day, and open source products to solve handles these needs like Elasticsearch became big business.
- Web frameworks for both front-end and back-end made it significant faster to develop web applications.

In addition to above technological changes, it also started to become more and more difficult to adapt Invenio v1 to all the different use cases we wanted to support. Preservation archives have vastly different requirements from aggregators which have vastly different requirements from research data management systems. We further started to see performance problems with larger and larger number of records.

Last but not least, we had many uses cases where it was no longer beneficial to store the records in MARC21, but instead adopt either newer or custom data model.

All in all, new technologies, an aging product showing its cracks, slow development and a wish to have other data models was key determining factors in deciding to start from scratch and implement a framework rather than a software application.

8.2 What happened to Invenio v2?

Initial in 2011 we started out on creating a hybrid application which would allow us to progressively migrate features as we had the time. In 2013 we launched Zenodo as the first site on the v2 development version which among other things featured Jinja templates instead of the previous Python based templates.

In theory everything was sound, however over the following years it became very difficult to manage the inflow of changes from larger and larger teams on the development side and operationally proved to be quite unstable compared to v1.

Last but not least, Invenio v1 was built in a time where the primary need was publication repositories and v2 inherited this legacy making it difficult to deal with very large research datasets.

Thus, in late 2015 we were being slowed so much down by our past legacy that we saw no other way that starting over from scratch if we were to deal with the next 20 years of challenges.

Notes on getting involved, contributing, legal information and release notes are here for the interested.

9.1 Maintenance Policy

Our goal is to ensure that all Invenio releases are supported with bug and security fixes for minimum one year after the release date and possibly longer. We further aim at one Invenio release with new features every 6 months. We strive our best to ensure that upgrades between minor versions are fairly straight-forward to ensure users follow our latest releases.

The maintenance policy is striving to strike a balance between maintaining a rock solid secure framework while ensuring that users migrate to latest releases and ensuring that we have enough resources to support the maintenance policy.

9.1.1 Types of releases

Major release: Major versions such as `v3` allows us to introduce major new features and make significant backward incompatible changes.

Minor releases: Minor versions such as `v3.1` allows us to introduce new features, make minor backward incompatible changes and remove deprecated features in a progressive manner.

Patch releases: Patch versions such as `v3.0.1` allows us fix bugs and security issues in a manner that allow users to upgrade immediately without breaking backward compatibility.

9.1.2 Policy

A minor release `A.B` (e.g. `v3.0`) is supported with bug and security fixes (via patch releases) until the release of `A.B+2` (e.g. `v3.2`) and minimum one year.

We may make exceptions to this policy for very serious security bugs.

9.1.3 End of life dates

Release	EOL Date	Maintained until
v3.0.1	2019-06-07	v3.2.0
v3.0.0	2019-06-07	v3.2.0
v2.x.y	2018-06-07	v3.0.0
v1.x.y	2018-06-07	v3.0.0

9.2 Version 3.0.1

Invenio v3.0.1 fixes several bugs in Invenio v3.0.0.

9.2.1 Bug fixes

- **Invenio-Access:** Fixed bug with the `any_user`-need not being provided by the anonymous identity.
- **Invenio-App:** Fixed compatibility issue with latest release of Flask-Talisman.
- **Invenio-DB:** Fixed a compatibility issue with newly released SQLAlchemy-Continuum.
- **Invenio-Search:** Fixed a compatibility issue with `elasticsearch-dsl>=6.2.0`.
- **Pytest-Invenio:** Fixed a problem which caused Invenio modules to be loaded despite the related test fixtures not being used, thus causing import errors if the module was not installed.
- **Pytest-Invenio:** Fixed a compatibility issue with `pytest v3.8.1+`.
- **Pytest-Invenio:** Fixed a problem with the Content-Security-Policy (CSP) in the default test application.

9.2.2 Enhancements

- **Invenio:** Added documentation section on building data models.
- **Invenio-App:** Added more relaxed Content Security Policy (CSP) when in DEBUG mode to allow usage of Flask-DebugToolbar.
- **Invenio-App:** Added support for loading multiple configuration modules from the `invenio_config` module-entry point group instead of only one module. The configuration modules are loaded in alphabetical ascending order based on the entry point name.
- **Invenio-Indexer:** Fixed issue that prevented configuration of the bulk indexing parameters sent to Elasticsearch.
- **Invenio-Records-REST:** Added support for permission checking on the search REST API endpoint (e.g. `/api/records/`).
- **Invenio-Search:** Added a new configuration variable `SEARCH_RESULTS_MIN_SCORE` to control minimum score needed in order to include a document in a result set.
- **Invenio-Search:** Added a new configuration variable `SEARCH_CLIENT_CONFIG` to allow control over Elasticsearch connection properties such as timeout and connection class.

9.2.3 Maintenance policy

Invenio v3.0 will be supported with bug and security fixes until the release of Invenio v3.2 and minimum until 2019-06-07.

9.3 Version 3.0.0

We are proud to announce the release of Invenio v3.0.0. Invenio has been completely rewritten from scratch with a radically improved architecture and technical implementation. Invenio 3 is now a framework, like a Swiss Army knife, complete with battle-tested, safe and secure modules providing all the features you need to build and run a trusted digital repository.

Whilst Invenio 3 is officially released to the world today, in reality it has already been relied upon in large-scale production systems for more than 1.5 years on sites such as:

- [Zenodo](#)
- [CERN Open Data](#)
- [CDS Videos](#)

Others sites are already in process of being built on Invenio 3:

- [INSPIRE HEP](#) - an aggregator for High-Energy Physics.
- [WEKO3](#) - repository platform for 500+ Japanese universities.

9.3.1 What's new

Invenio functionality is being released in **bundles** of modules. Invenio v3 contains the following bundles totaling more than 27 individual Invenio modules:

- **Base:** the core application framework with e.g. distributed task queue support.
- **Auth:** accounts management, role-based access control, OAuth 2.0 client and provider, user profiles management.
- **Metadata:** record and persistent identifier management including indexing, querying and OAI-PMH server.

The following bundles are being prepared for release in v3.1:

- **Files:** advanced file management with multi-backend support as well as IIIF Image API support.
- **Statistics:** [COUNTER-compliant](#) statistics.

See our [roadmap](#) for further details.

9.3.2 Getting started

In order to get started developing with Invenio v3 follow our [Launch an Invenio instance](#).

Next, head over <https://invenio.readthedocs.io> to understand how to develop with Invenio.

In addition, each Invenio module also has extensive documentation:

Base bundle

- [invenio-admin](#)
- [invenio-app](#)
- [invenio-assets](#)
- [invenio-base](#)
- [invenio-celery](#)
- [invenio-config](#)

- invenio-db
- invenio-formatter
- invenio-i18n
- invenio-logging
- invenio-mail
- invenio-rest
- invenio-theme

Auth bundle

- invenio-access
- invenio-accounts
- invenio-oauth2server
- invenio-oauthclient
- invenio-userprofiles

Metadata bundle

- invenio-indexer
- invenio-jsonschemas
- invenio-oaiserver
- invenio-pidstore
- invenio-records
- invenio-records-rest
- invenio-records-ui
- invenio-search
- invenio-search-ui

9.3.3 Compatibilities

Python compatibility

Invenio v3.0 supports Python 2.7, 3.5, 3.6. We highly recommend only using the latest official release in each series.

Python 2.7 end-of-life is scheduled for April 2020. Invenio will only support Python 2.7 until that date. We highly recommend that all new projects are started latest available Python 3 version.

Elasticsearch compatibility

Invenio v3.0 supports Elasticsearch 2, 5 and 6.

Elasticsearch v2 has reached end-of-life (February 2018) and Invenio v3.0 is the last release to support Elasticsearch v2.

PostgreSQL compatibility

Invenio v3.0 supports PostgreSQL 9.4, 9.5 and 9.6. We have not yet tested Invenio v3.0 with PostgreSQL 10.

MySQL compatibility

Invenio v3.0 supports MySQL 5.6+.

9.3.4 Deprecations

AMD/RequireJS

Invenio v3.0's current static assets management system is based on e.g. RequireJS will be replaced with Webpack. We expect this work to be ready for Invenio v3.1, and thus we are already deprecating the current support. Specifically this means that Invenio-Assets and Invenio-Theme will change significantly in Invenio v3.1. We would have liked to already have this ready for this v3.0 release, but unfortunately it was time-wise not possible.

AngularJS

Invenio v3.0 comes with one AngularJS 1.4 application (Invenio-Search-JS). AngularJS is by now already outdated, and we are planning a rewrite of the application in another JavaScript framework that is currently in process of being selected. Essentially this means that you should not extend Invenio-Search-JS at this point, since it will change significantly.

9.3.5 Maintenance policy

Invenio v3.0 will be supported with bug and security fixes until the release of Invenio v3.2 and minimum until 2019-06-07.

We aim at one Invenio release with new features every 6 months. We expect upgrades between minor versions (e.g. v3.1 to v3.2) to be fairly straight-forward as in most cases only new features are added.

9.4 Version 2.x

End-of-life

Invenio v2.x code base is a hybrid architecture that uses Flask web development framework combined with Invenio v1.x framework.

Note: The 2.x code base is not suitable for production systems, and will not receive any further development nor security fixes.

Released versions include:

Invenio v2.1:

- v2.1.1 - released 2015-09-01
- v2.1.0 - released 2015-06-16

Invenio v2.0:

- [v2.0.6](#) - released 2015-09-01
- [v2.0.5](#) - released 2015-07-17
- [v2.0.4](#) - released 2015-06-01
- [v2.0.3](#) - released 2015-05-15
- [v2.0.2](#) - released 2015-04-17
- [v2.0.1](#) - released 2015-03-20
- [v2.0.0](#) - released 2015-03-04

9.5 Version 1.x

End-of-life

Invenio v1.x code base is suitable for stable production. It uses legacy technology and custom web development framework.

Note: Invenio v1.x has reached end-of-life and will not receive any further development nor security fixes.

Invenio v1.2.2 is the last stable release from the old Invenio legacy technology code base. If you have been using one of previous Invenio versions, it is recommended to upgrade to this version.

Note: If you would like to check out Invenio v1.2 locally please see our important [note](#) how to install it.

Released versions include:

Invenio v1.2:

- [v1.2.2](#) - released 2016-11-25
- [v1.2.1](#) - released 2015-05-21
- [v1.2.0](#) - released 2015-03-03

Invenio v1.1:

- [v1.1.7](#) - released 2016-11-20
- [v1.1.6](#) - released 2015-05-21
- [v1.1.5](#) - released 2015-03-02
- [v1.1.4](#) - released 2014-08-31
- [v1.1.3](#) - released 2014-02-25
- [v1.1.2](#) - released 2013-08-19
- [v1.1.1](#) - released 2012-12-21
- [v1.1.0](#) - released 2012-10-21

Invenio v1.0:

- [v1.0.10](#) - released 2016-11-09
- [v1.0.9](#) - released 2015-05-21

- [v1.0.8](#) - released 2015-03-02
- [v1.0.7](#) - released 2014-08-31
- [v1.0.6](#) - released 2014-01-31
- [v1.0.5](#) - released 2013-08-19
- [v1.0.4](#) - released 2012-12-21
- [v1.0.3](#) - released 2012-12-19
- [v1.0.2](#) - released 2012-10-19
- [v1.0.1](#) - released 2012-06-28
- [v1.0.0](#) - released 2012-02-29
- [v1.0.0-rc0](#) - released 2010-12-21

9.6 Version 0.x

Invenio v0.x code base was developed and used in production instances since 2002. The code base is interesting only for archaeological purposes.

Released versions include:

- [v0.99.9](#) - released 2014-01-31
- [v0.99.8](#) - released 2013-08-19
- [v0.99.7](#) - released 2012-12-18
- [v0.99.6](#) - released 2012-10-18
- [v0.99.5](#) - released 2012-02-21
- [v0.99.4](#) - released 2011-12-19
- [v0.99.3](#) - released 2010-12-13
- [v0.99.2](#) - released 2010-10-20
- [v0.99.1](#) - released 2008-07-10
- [v0.99.0](#) - released 2008-03-27
- [v0.92.1](#) - released 2007-02-20
- [v0.92.0](#) - released 2006-12-22
- [v0.90.1](#) - released 2006-07-23
- [v0.90.0](#) - released 2006-06-30
- [v0.7.1](#) - released 2005-05-04
- [v0.7.0](#) - released 2005-04-06
- [v0.5.0](#) - released 2004-12-17
- [v0.3.3](#) - released 2004-07-16
- [v0.3.2](#) - released 2004-05-12
- [v0.3.1](#) - released 2004-03-12
- [v0.3.0](#) - released 2004-03-05

- [v0.1.2](#) - released 2003-12-21
- [v0.1.1](#) - released 2003-12-19
- [v0.1.0](#) - released 2003-12-04
- [v0.0.9](#) - released 2002-08-01

Notes on getting involved, contributing, legal information and release notes are here for the interested.

10.1 Getting help

Didn't find a solution to your problem the Invenio documentation? Here's how you can get in touch with other users and developers:

Forum/Knowledge base

- <https://github.com/inveniosoftware/troubleshooting>

Ask questions or browse answers to existing questions.

Chatroom

- <https://gitter.im/inveniosoftware/invenio>

Probably the fastest way to get a reply is to join our chatroom. Here most developers and maintainers of Invenio hangout during their regular working hours.

GitHub

- <https://github.com/inveniosoftware>

If you have feature requests or want to report potential bug, you can do it by opening an issue in one of the individual Invenio module repositories. In each repository there is a MAINTAINERS file in the root, which lists the who is maintaining the module.

10.2 Communication channels

10.2.1 Chatrooms

Most day-to-day communication is happening in our chatrooms:

- [Public chatroom](#) (for everyone)
- [Developer chatroom](#) (for [members](#) of inveniosoftware GitHub organisation).

If you want to join the developer chatroom, just ask for an invite on the public chatroom.

10.2.2 GitHub

Most of the developer communication is happening on GitHub. You are strongly encouraged to join <https://github.com/orgs/inveniosoftware/teams/developers> team and watch notifications from various <https://github.com/inveniosoftware/> organisation repositories of interest.

10.2.3 Meetings

Invenio Developer Forum

Monday afternoons at 16:00 CET/CEST time we meet physically at CERN and virtually over videoconference to discuss interesting development topics.

You can join our Monday forums from anywhere via videoconference. Here the steps:

- View the [agenda](#) ([ical feed](#)).
- Install the videoconferencing client [Vidyo](#).
- Join our [virtual room](#).

Invenio User Group Workshop

We meet among Invenio users and developers from around the world at a yearly Invenio User Group Workshop. The workshop consists of a series of presentations, tutorials, practical exercises, and discussions on topics related to Invenio digital library management and development. We exchange knowledge and experiences and drive forward the forthcoming developments of the Invenio platform.

See list of [upcoming](#) and [past](#) workshops.

Other meetings

In addition to above meetings where everyone can join, the following meetings exists:

- Quarterly project coordination meeting with the Invenio project coordinators and architects that help steer the project prioritizes and goals.
- Weekly architecture meeting.

10.2.4 Project website

Our project website, <http://inveniosoftware.org>, is used to show case Invenio.

10.2.5 Email

You can get in touch with the Invenio management team on info@inveniosoftware.org.

In particular use above email address to report security related issues privately to us, so we can distribute a security patch before potential attackers look at the issue.

10.2.6 Twitter

The official Twitter account for Invenio software is used mostly for announcing new releases and talks at conferences:

- <https://twitter.com/inveniosoftware>

10.2.7 Mailing lists

The mailing lists are currently not very active and are primarily listed here for historical reasons.

- `project-invenio-announce@cern.ch`: Read-only moderated mailing list to announce new Invenio releases and other major news concerning the project. [subscribe to announce](#), [archive](#)
- `project-invenio-general@cern.ch`: Originally used for discussion among users and administrators of Invenio instances. The mailing list has mostly been replaced by our public chatroom. [subscribe to general](#), [new general archive](#), [old general archive](#)
- `project-invenio-devel@cern.ch`: Originally used for discussion among Invenio developers. The mailing list has mostly been replaced by our developer chatroom. [subscribe to devel](#), [new devel archive](#), [old devel archive](#)

Note that all the mailing lists are also archived (as of the 20th of July, 2011) on [The Mail Archive](#).

10.3 Contribution guide

Interested in contributing to the Invenio project? There are lots of ways to help.

Code of conduct

Overall, we're **open, considerate, respectful and good to each other**. We contribute to this community not because we have to, but because we want to. If we remember that, our *Code of Conduct* will come naturally.

Get in touch

See *Getting help* and *Communication channels*. Don't hesitate to get in touch with the Invenio maintainers. The maintainers can help you kick start your contribution.

10.3.1 Types of contributions

Report bugs

- **Found a bug? Want a new feature?** Open a GitHub issue on the applicable repository and get the conversation started (do search if the issue has already been reported). Not sure exactly where, how, or what to do? See *Getting help*.

- **Found a security issue?** Alert us privately at info@inveniosoftware.org, this will allow us to distribute a security patch before potential attackers look at the issue.

Translate

- **Missing your favourite language?** Translate Invenio on [Transifex](#)
- **Missing context for a text string?** Add context notes to translation strings or report the issue as a bug (see above).
- **Need help getting started?** See our *Translation guide*.

Write documentation

- **Found a typo?** You can edit the file and submit a pull request directly on GitHub.
- **Debugged something for hours?** Spare others time by writing up a short troubleshooting piece on <https://github.com/inveniosoftware/troubleshooting/>.
- **Wished you knew earlier what you know now?** Help write both non-technical and technical topical guides.

Write code

- **Need help getting started?** See our *Launch an Invenio instance*.
- **Need help setting up your editor?** See our *Developer environment guide* guide which helps your automate the tedious tasks.
- **Want to refactor APIs?** Get in touch with the maintainers and get the conversation started.
- **Troubles getting green light on Travis?** Be sure to check our *Style guide* and the *Developer environment guide*. It will make your contributor life easier.
- **Bootstrapping a new awesome module?** Use our Invenio cookiecutter templates for [modules](#), [instances](#) or [data models](#)

10.3.2 Style guide (TL;DR)

Travis CI is our style police officer who will check your pull request against most of our *Style guide*, so do make sure you get a green light from him.

ProTip: Make sure your editor is setup to do checking, linting, static analysis etc. so you don't have to think. Need help setting up your editor? See *Developer environment guide*.

Commit messages

Commit message is first and foremost about the content. You are communicating with fellow developers, so be clear and brief.

(Inspired by [How to Write a Git Commit Message](#))

1. Separate subject from body with a blank line
2. Limit the subject line to 50 characters
3. Indicate the component follow by a short description

4. Do not end the subject line with a period
5. Use the imperative mood in the subject line
6. Wrap the body at 72 characters
7. Use the body to explain what and why vs. how, using bullet points

ProTip: Really! Spend some time to ensure your editor is top tuned. It will pay off many-fold in the long run. See [Developer environment guide](#).

For example:

```
component: summarize changes in 50 char or less

* More detailed explanatory text, if necessary. Formatted using
  bullet points, preferably `*`. Wrapped to 72 characters.

* Explain the problem that this commit is solving. Focus on why you
  are making this change as opposed to how (the code explains that).
  Are there side effects or other unintuitive consequences of this
  change? Here's the place to explain them.

* The blank line separating the summary from the body is critical
  (unless you omit the body entirely); various tools like `log`,
  `shortlog` and `rebase` can get confused if you run the two
  together.

* Use words like "Adds", "Fixes" or "Breaks" in the listed bullets to help
  others understand what you did.

* If your commit closes or addresses an issue, you can mention
  it in any of the bullets after the dot. (closes #XXX) (addresses
  #YYY)

Co-authored-by: John Doe <john.doe@example.com>
```

Git signature: The only signature we use is Co-authored-by (see above) to provide credit to co-authors. Previously we required a Signed-off-by signature, however this is no longer required.

10.3.3 Pull requests

Need help making your first pull request? Check out the GitHub guide [Forking Projects](#).

When making your pull request, please keep the following in mind:

- Create logically separate commits for logically separate things.
- Include tests and don't decrease test coverage.
- Do write documentation. We all love well-documented frameworks, right?
- Run tests locally using `run-tests.sh` script.
- Make sure you have the rights if you include third-party code (and do credit the original creator). Note, you cannot include GPL or AGPL licensed code. LGPL and other more permissive open source license or fine.
- Green light on all GitHub status checks is required in order to merge your PR.

Work in progress (WIP)

Do publish your code as pull request sooner than later. Just prefix the pull request title with WIP (=work in progress) if it is not quite ready.

Allow edits from maintainers

To speed up the integration process, it helps if on GitHub you [allow maintainers to edit your pull request](#) so they can fix small issues autonomously.

10.4 Style guide

This style guide can to a large degree be fully automated by your editor. See our [Developer environment guide](#) for help on how to setup your editor.

10.4.1 Python

We follow [PEP-8](#) and [PEP-257](#) and sort imports via `isort`. Please plug corresponding linters such as `flake8` to your editor.

10.4.2 Indention style and whitespace

Each repository contains a `.editorconfig` that defines the indention style, character set, trimming of trailing whitespace and final new lines.

10.5 Developer environment guide

You can save a lot of time and frustrations by spending some time setting up your development environment. We have primarily adopted existing community style guides, and in most cases the formatting and checking can be fully automated by your editor.

10.5.1 Editor

You can use any code editor of your choice. Here we give a brief overview of some of the editors our existing developers are using. For all editors, the most important is support for [EditorConfig](#)

EditorConfig

All repositories have a `.editorconfig` file which defines indention style, text encoding, newlines etc. Many editors either come with built-in support or plugins that reads the `.editorconfig` file and configures your editor accordingly.

See [EditorConfig](#) for list of supported editors.

Editors

Following editors (listed alphabetically) are used by our existing developers. Don't hesitate to reach out on our Gitter channel, to ask for help for useful plugins:

- [Atom](#)
- [Emacs](#)
- [PyCharm](#)
- [Sublime](#)
- [VIM](#)
- [Visual Studio Code](#)

Plugins for editors

The key plugins you should look for in your editor of choice are:

- [Python / JavaScript](#) environment
- [PEP8 / PEP257](#) style checking
- [Isort](#) plugin.

10.6 Translation guide

Invenio has been translated to more than 25 languages.

10.6.1 Transifex

All Invenio internationalisation and localisation efforts are happening on the [Transifex](#) collaborative localisation platform.

You can start by exploring [Invenio project dashboard](#) on Transifex.

Invenio project consists of many resource files which can be categorised as follows:

- `invenio-maint10-messages` contains phrases for Invenio 1.0 legacy release series
- `invenio-maint11-messages` contains phrases for Invenio 1.1 legacy release series
- `invenio-maint12-messages` contains phrases for Invenio 1.2 legacy release series
- `invenio-xxx-messages` contains phrases for Invenio 3.0 `invenio-xxx` module

We follow the usual Transifex localisation workflow that is [extensively documented](#).

10.6.2 Translators

All contributions with translating phrases or with reviewing translations are very appreciated!

Please read [Getting started as a translator](#) and join Invenio project on Transifex.

10.6.3 Developers

Please see dedicated [invenio-i18n](#) documentation.

10.7 Maintainer's guide

Following is a guide for *maintainers* and *architects* who are the responsible for maintaining one or more repositories.

10.7.1 Overview

The goal of the maintainer system is to share the load of maintaining Invenio, spread Invenio expertise and mentor new contributors. Overall maintainers are key to ensuring that Invenio has a welcoming, responsive, collaborative open and transparent community.

The maintainer system is designed to allow contributors to progressively take more and more responsibility in the project, while allowing for mistakes and providing contributors with training, support and mentorship they need in order to perform their tasks.

What is a maintainer?

Maintainers are first and foremost service people. They help drive Invenio development forward and ensure newcomers as well as long-time contributors have a great experience when contributing to Invenio, which is key to Invenio's longterm success.

Maintainers are not only developers. We also need maintainers for translations, the website, forums, interest groups etc.

Taks:

- Help contributors get their contributions integrated in Invenio.
- Answer questions and help users in e.g. the chat rooms (see [Communication channels](#)).
- Manage issues, assignments and milestones.
- Review and merge pull requests.
- Prepare releases.
- Participate in feature development, bug fixing and proactively verifies if nightly builds are failing.
- Be the domain expert on a specific topic (e.g. know the architecture of a module and how it fits into the rest of Invenio).
- Help scout for and train for potential new maintainers.

What is an architect?

Architects are like normal maintainers, they just maintain many more repositories and have larger overview over the Invenio architecture and ecosystem. Architects shapes and drives the overall Invenio technical architecture.

Tasks (in addition to maintainer tasks):

- Provide mentorship for maintainers.
- Determine the overall Invenio architecture and ecosystem and ensure coherence throughout the Invenio development.

- Be the Invenio experts and know of use cases and interdependencies between different modules.
- Signs off on new releases in their respective repositories.

Becoming a maintainer

Don't forget: being a maintainer is a time investment. Make sure you will have time to make yourself available. You don't have to be a maintainer to make a difference on the project!

Still sounds like something for you? Get in contact with one the Invenio product manager, one of the architects or one of the project coordinators. They will help you through the process. See [Communication channels](#).

Stepping down as maintainer

Don't have enough time? Changing jobs? No problem, all we ask is that you step down gracefully. Try to help find a new maintainer (if you don't already have a co-maintainer) and help hand-over pending tasks and knowledge to the existing or new maintainers.

As soon as you know you want to step down, please notify the Invenio product manager or one of the architects team members so we can help in the transition.

10.7.2 Setting up a repository

First, reach out to the Invenio architects and agree on scope and name of the new repository. The architects are there to help you and to ensure that the module fits into the larger Invenio ecosystem.

New repositories can be created in either the [inveniosoftware](#) or the [inveniosoftware-contrib](#) GitHub organisations. Repositories in [inveniosoftware](#) must be managed according to the contributor, style and maintainers guides. Repositories in [inveniosoftware-contrib](#) are free to apply any rules they like.

GitHub

Once scope and name has been agreed upon, the product manager will create the repository which will be setup in the following manner:

- **Settings: The repository settings must be set in the following manner:**
 - Description and homepage (link to [readthedocs.io](#)) *must* be set.
 - Issues *must* be enabled.
 - Wiki *should* be disabled (except in rare circumstances such as the main Invenio repository).
 - Merge button: *must* disallow merge commits, allow squash and allow rebase merging.
- **Teams:** A team named `<repository-name>-maintainers` *must* exist with all repository maintainers as members and with `push` permission on the repository.
- **Branch protection:** The default branch, all maintenance branches and optionally some feature branches *must* have branch protection enabled in the following manner:
 - Pull requests reviews *must not* be required. Enabling this feature prevents maintainers from merging their own PRs without approval from another reviewer. This is not a carte blanche for maintainers to merge their own PRs without reviews, but empowers them to get the job done when really need!
 - Status checks for TravisCI *must* be required. Status checks for Coveralls, QuantifiedCode and other status checks *must not* be required. The maintainer is responsible for manually verifying these checks.

- Branches *must* be up to date before merging.
- Push access *must* be restricted to the repository maintainers team.
- **Repository files:** A `MAINTAINERS` file with list of GitHub usernames *must* be present in the repository.

The repository setup and manage is fully automated via the [MetaInvenio](#) scripts.

Other services

We use the following other external services:

- [TravisCI](#) for continues integration testing.
- [Coveralls](#) to test coverage tracking.
- [QuantifiedCode](#) for Python static analysis.
- [Python Package Index](#) for releasing Python packages.
- [NPM](#) for releasing JavaScript packags.
- [ReadTheDocs](#) for hosting documentation.
- [Transifex](#) for translating Invenio.

Bootstrapping

New repositories should in most cases be bootstrapped using one of our templates. These templates encodes many best practices, setups above external services correctly and ensure a coherent package structure throughout the Invenio project.

Python

Python-based repositories must be bootstrapped using the [cookiecutter-invenio-module](#).

JavaScript

JavaScript-based repositories must be bootstrapped using the [generator-invenio-js-module](#).

10.8 Governance

Invenio is governed by CERN for the benefit of the community. CERN strives to make Invenio a collaborative, open and transparent project to ensure that everyone can contribute and have their say on the directions of the project.

Invenio governance is in general informal and we try to strike a balance between processes and agreed upon standards vs. the wild west where everyone do as they see fit. The governance model is intended to allow that people progressively take larger and larger responsibilities with support from the existing leadership.

These following sections define the different roles and responsibilities in the project, define how decisions are taken and how people are appointed to different roles. Overall, we expect every person who participates in the project to adhere to our [Code of Conduct](#).

Note: The current governance model puts in words how the collaboration currently works in practice today and sets a basic framework for how we collaborate and take decisions in the project.

If the nature of the community or contributors changes this governance model may be reviewed and changed if necessary.

10.8.1 Roles and responsibilities

The *product manager*, *coordinators*, *architects* and *maintainers* (as defined below) make up the leadership of Invenio. The leaders of Invenio are **service people** who:

- take an active role in driving the project forward,
- help newcomers as well as long-time contributors have great experience contributing to Invenio,
- help train members to progressively take larger responsibility in the project,
- are role models for the remaining community.

Roles:

- **Members:** Anyone using Invenio.
- **Contributors:** Anyone contributing to Invenio (in it widest possible interpretation, i.e. not only programmers).
- **Maintainers:** Anyone maintaining at least one repository. Maintainers are responsible for managing the issues and/or the code base of a repository according to Invenio's standards.
- **Architects:** Anyone maintaining 20+ repositories (though max 10 people). Architects are responsible for the overall Invenio technical architecture as well as managing and training maintainers on their respective repositories.
- **Coordinators:** Representatives of Invenio based services that would like to coordinate their Invenio development efforts with other services and provide input on the product road map.
- **Product manager:** Overall responsible for Invenio's vision, strategy and day-to-day management. Responsible for managing and training architects and coordinators.

Commit access on repositories are given to contributors, maintainers and architects. Contributors can commit/merge to feature branches while only maintainers and architects can commit/merge to master/maintenance branches (meaning also only they can release packages to PyPI and NPM).

10.8.2 Decision making

We strive to take decisions openly and by consensus, though ultimately CERN represented by the Invenio product manager has the final say on all decisions in the project. In particular this means that there is no formal voting procedure for Invenio.

Leaders drive decision making

The Invenio product manager, architects, coordinators and maintainers as the leaders of the project are responsible for driving decision making in their respective domains.

Driving decision making means:

- facilitating an open constructive discussion around a decision that matches the level of importance and impact of a decision,

- striving for reaching consensus on a decision and ensuring relevant other members are aware and included on the decision,
- ensuring decisions are in alignment with the overall Invenio vision, strategy, architecture and standards,
- coordinating the decision with the Invenio leadership (product manager, architects and coordinators),
- taking the decision.

Leaders implement decisions

Leaders are responsible for following up decisions they take by actual implementation. Decision should not be considered final unless it is actually implemented or documented publicly.

Disagreements

Leaders of the project should always strive for consensus. If that is not possible the leader taking a decision should alert the Invenio product manager prior to taking the decision.

Members who are disagreeing with a decision may ask the product manager to review a specific decision and possible change it.

Members who are disagreeing with the Invenio product manager may escalate the product manager's decision to their hierarchy at CERN.

10.8.3 Appointments

The Invenio product manager is appointed by CERN. Architects, coordinators and maintainers are appointed by the Invenio product manager in collaboration with existing architects and coordinators.

Maintainers are appointed by the architects (e.g. a new Invenio module) or coordinators (e.g. a new special interest group).

In general, appointments are made in an informal way, and usually anyone volunteering that have been showing commitment to the project will get appointed. Any member can volunteer or suggest other members for roles.

Revoking of appointed roles

The product manager may revoke appointed roles of a member for reasons such as (but not limited to):

- lack of activity
- violations of the code of conduct
- repeated infringements of the contribution, style or maintainer guides.

The product manager must give a warning to the member to allow them to correct their behavior except in severe cases. Revoking roles should be a last measure, and only serve the purpose to ensure that Invenio has a healthy community and collaboration based on our [Code of Conduct](#).

10.8.4 Working/Interest groups

Working/interest groups may be set up by the product manager on request of any group of members who wish to address a particular area of Invenio (say MARC21 support or research data management). Working/interest groups help coordinate the overall vision, strategy and architecture of a specific area of Invenio. Each working/interest group must have chair that reports to the product manager.

10.9 Code of Conduct

We endorse the [Python Community Code of Conduct](#):

The Invenio community is made up of members from around the globe with a diverse set of skills, personalities, and experiences. It is through these differences that our community experiences great successes and continued growth. When you're working with members of the community, we encourage you to follow these guidelines which help steer our interactions and strive to keep Invenio a positive, successful, and growing community.

A member of the Invenio community is:

1. **Open.** Members of the community are open to collaboration, whether it's on RFCs, patches, problems, or otherwise. We're receptive to constructive comment and criticism, as the experiences and skill sets of other members contribute to the whole of our efforts. We're accepting of all who wish to take part in our activities, fostering an environment where anyone can participate and everyone can make a difference.
2. **Considerate.** Members of the community are considerate of their peers – other Invenio users. We're thoughtful when addressing the efforts of others, keeping in mind that often times the labor was completed simply for the good of the community. We're attentive in our communications, whether in person or online, and we're tactful when approaching differing views.
3. **Respectful.** Members of the community are respectful. We're respectful of others, their positions, their skills, their commitments, and their efforts. We're respectful of the volunteer efforts that permeate the Invenio community. We're respectful of the processes set forth in the community, and we work within them. When we disagree, we are courteous in raising our issues.

Overall, we're good to each other. We contribute to this community not because we have to, but because we want to. If we remember that, these guidelines will come naturally.

We recommend the “egoless” programming principles (Gerald Weinberg, [The Psychology of Computer Programming](#), 1971):

1. **Understand and accept that you will make mistakes.** The point is to find them early, before they make it into production. Fortunately, except for the few of us developing rocket guidance software at JPL, mistakes are rarely fatal in our industry, so we can, and should, learn, laugh, and move on.
2. **You are not your code.** Remember that the entire point of a review is to find problems, and problems will be found. Don't take it personally when one is uncovered.
3. **No matter how much “karate” you know, someone else will always know more.** Such an individual can teach you some new moves if you ask. Seek and accept input from others, especially when you think it's not needed.
4. **Don't rewrite code without consultation.** There's a fine line between “fixing code” and “rewriting code.” Know the difference, and pursue stylistic changes within the framework of a code review, not as a lone enforcer.
5. **Treat people who know less than you with respect, deference, and patience.** Nontechnical people who deal with developers on a regular basis almost universally hold the opinion that we are prima donnas at best and crybabies at worst. Don't reinforce this stereotype with anger and impatience.
6. **The only constant in the world is change.** Be open to it and accept it with a smile. Look at each change to your requirements, platform, or tool as a new challenge, not as some serious inconvenience to be fought.
7. **The only true authority stems from knowledge, not from position.** Knowledge engenders authority, and authority engenders respect – so if you want respect in an egoless environment, cultivate knowledge.
8. **Fight for what you believe, but gracefully accept defeat.** Understand that sometimes your ideas will be overruled. Even if you do turn out to be right, don't take revenge or say, “I told you so” more than a few times at most, and don't make your dearly departed idea a martyr or rallying cry.

9. **Don't be “the guy in the room”.** Don't be the guy coding in the dark office emerging only to buy cola. The guy in the room is out of touch, out of sight, and out of control and has no place in an open, collaborative environment.
10. **Critique code instead of people** – be kind to the coder, not to the code. As much as possible, make all of your comments positive and oriented to improving the code. Relate comments to local standards, program specs, increased performance, etc.

10.10 License

MIT License

Copyright (C) 2015-2018 CERN.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Note: In applying this license, CERN does not waive the privileges and immunities granted to it by virtue of its status as an Intergovernmental Organization or submit itself to any jurisdiction.
