invenio Documentation

Release 3.1.2

CERN

Feb 11, 2020

Contents

Open Source framework for large-scale digital repositories.

Invenio is like a Swiss Army knife of battle-tested, safe and secure modules providing you with all the features you need to run a trusted digital repository.

CHAPTER 1

Overview

1.1 Introduction

Invenio is an open source framework for building large-scale digital repositories.

1.1.1 A framework

Invenio is first and foremost a framework. It's a framework that you can use to build a turn-key repository solution, but it is not by itself a final turn-key repository software. Using Invenio requires you to develop code.

Several repositories have been built on top of Invenio v3, including e.g.:

- Zenodo - General purpose research data repository.
- CERN Open Data - Open data repository for CERN.
- CERN Videos - Digital Assets Management system with video encoding support.

Several repository software solutions are in the progress of being written on top of Invenio, including e.g.:

- RERO ILS and Invenio ILS - Integrated Library System.
- WEKO3 - Repository infrastructure for 500+ japanese universities.

1.1.2 Scalability & Safety

Two of the main strengths of Invenio is the scalability and safety. Invenio is built to run on anything from a single machine to clusters of 100s of machines, to handle 100 records or 100 million records as well as to handle a 1 megabyte or a 1 petabyte.

That's why we say Invenio is a framework for large-scale digital repositories. Often, large-scale repositories does not fit in a standard box, which is why Invenio is first and foremost a framework that helps you build your repository faster and on a high-quality reliable foundation.

1.1.3 Flexible metadata

In its core, Invenio provides you with a flexible record and persistent identifier store capable of handling 100 millions of records. Records can use existing metadata formats such as JSON-LD, MARC21, DublinCore, DataCite, as well as your own custom or derived metadata format. Invenio easily handle multiple types of records such as bibliographic records, authority records, people, grants, funders, books and photos to name a few.

Internally Invenio natively store records as JSON documents whose structure can be validated and described with JSONSchemas. Records can easily be linked via JSONRef providing you with powerful tools to model your records. Invenio further comes with robust metadata transformation layer that can serialize records to e.g. MARCXML, DataCite XML, JSON-LD, Citation Style Langauge (CSL) JSON and many other formats.

In addition Invenio provides a persistent identifier store and a resolver that allows you to use your preferred persistent identifier scheme for identifying records such as DOIs (Digital Object Identifiers), Handles, PURLs, URNs or your own local identifier. The persistent identifer resolver further has support for advanced features such as tombstone pages, redirection and merged records.

1.1.4 Powerful search

Under the hood, Invenio uses Elasticsearch, the world's most popular open source search engine that provides powerful distributed and massively scalable search engine. Invenio provides all the features of Elasticsearch such as full-text search, powerful query syntax, advanced stemming and aggregations, super-fast auto-completion suggesters as well as geospatial search.

Invenio further leverages both instant indexing as well as extremely fast distributed bulk indexing with rates beyond 10,0000 records/second.

1.1.5 File management

Invenio handles both millions of files and petabytes of data. Invenio provides natively on object storage REST API that concurrently can use multiple underlying storage systems such as S3, XRootD, NAS, WebDAV.

The file store further handles integrity checking according to desired schedules and supports even checking close to 1M files in less than an hour if your storage backend can support it.

The file store further comes with quota management that allows you detailed control of max file sizes and upload limits.

1.2 Bundles

Invenio is a highly modular framework with many modules that provide various different functionality. We are packing related modules into bundles which is released together at the same time.

Each module has a separate documentation which you can find linked below.

1.2.1 Base bundle

The base bundle contains all modules related to the generic web application. This includes the Flask/Celery application factories, configuration management, I18N, logging, database management, assets/theme management, mail handling and administration interface.

Included modules:

- **invenio-admin**

  - Administration interface for Invenio based on Flask-Admin.
- **invenio-app**
  - Flask, WSGI, Celery and CLI applications for Invenio including security-related headers and rate limiting.
- **invenio-assets**
  - Static files management and Webpack integration for Invenio.
- **invenio-base**
  - Flask application factories implementing the application loading patterns with entry points in Invenio.
- **invenio-cache**
  - Caching module for Invenio, supporting Reddis and Memcached as backends.
- **invenio-celery**
  - Task discovery and default configuration of Celery for Invenio.
- **invenio-config**
  - Configuration loading pattern responsible for loading configuration from Python modules, instance folder and environment variables.
- **invenio-db**
  - Database connection management for Invenio.
- **invenio-formatter**
  - Jinja template engine utilities for Invenio.
- **invenio-i18n**
  - I18N utilities like user locale detection, message catalog merging and views for language change.
- **invenio-logging**
  - Configuration of logging to both console, files and log aggregation engines like sentry.io
- **invenio-mail**
  - Mail sending for Invenio using Flask-Mail.
- **invenio-rest**
  - REST API utilities including Cross Origin Resource Sharing (CORS) and Content Negotiation versioning support.
- **invenio-theme**
  - Jinja templates implementing a basic theme for Invenio as well as menus and breadcrumbs support.
- **docker-invenio**
  - Docker base images based on CentOS 7 for Invenio.
- **pytest-invenio**
  - Testing utilities for Invenio modules and applications.

1.2.2 Auth bundle

The auth bundle contains all modules related to account and access management, user profiles, session management and OAuth (provider and client)

Included modules:

- **invenio-access**
  - Role Based Access Control (RBAC) with object level permissions.
- **invenio-accounts**
  - User/role management, registration, password recovery, email verification, session theft protection, strong cryptographic hashing of passwords, hash migration, session activity tracking and CSRF protection of REST API via JSON Web Tokens.
- **invenio-oauth2server**
  - OAuth 2.0 Provider for REST API authentication via access tokens.
- **invenio-oauthclient**
  - User identity management and support for login via ORCID, GitHub, Google or other OAuth providers.
- **invenio-userprofiles**
  - User profiles for integration into registration forms.

The modules relies heavily on a suite of open source community projects:

- flask-security
- flask-login
- flask-principal
- flask-oauthlib
- passlib

1.2.3 Metadata bundle

The metadata bundle contains all modules related to records and metadata management including e.g. records storage, persistent identifier management, search engine indexing, an OAI-PMH server and REST APIs for records.

Included modules:

- **invenio-indexer**
  - Highly scalable record bulk indexing.
- **invenio-jsonschemas**
  - JSONSchema registry for Invenio.
- **invenio-oaiserver**
  - OAI-PMH server implementation for Invenio.
- **invenio-pidstore**
  - Management, registration and resolution of persistent identifiers including e.g DOIs.
- **invenio-records**

  - JSON document storage with revision history and JSONSchema validation.
- **invenio-records-rest**
  - REST APIs for search and CRUD operations on records and persistent identifiers.
- **invenio-records-ui**
  - User interface for displaying records.
- **invenio-search**
  - Elasticsearch integration module for Invenio.
- **invenio-search-js**
  - AngularJS search application for displaying records from the REST API.
- **invenio-search-ui**
  - User interface for searching records.

1.2.4 Files bundle (beta)

Note: This bundle is in beta. The modules are being used in production systems but are still missing some minor changes as well as documentation.

The files bundle contains all modules related to management of files in Invenio, including an object storage REST API, multiple supported storage backends, file previewers, and IIIF image server and an integration layer between files and records.

Included modules:

- **invenio-files-rest**
  - Object storage REST API for Invenio with many supported backend storage protocols and file integrity checking.
- **invenio-iiif**
  - International Image Interoperability Framework (IIIF) server for making thumbnails and zooming images.
- **invenio-previewer**
  - Previewer for Markdown, JSON/XML, CSV, PDF, JPEG, PNG, TIFF, GIF and ZIP files.
- **invenio-records-files**
  - Integration layer between object storage and records.
- **invenio-xrootd**
  - Support for the storage protocol XRootD in Invenio.

1.2.5 Statistics bundle (beta)

Note: This bundle is in beta. The modules are being used in production systems but are still missing some minor changes as well as documentation.

The statistics bundle contains all modules related to counting statistics such as file downloads, record views or any other type of events. It supports the COUNTER Code of Practice as well as Making Data Count Code of Practice including e.g. double-click detection.

Included modules:

- **invenio-stats**
  - Event collection, processing and aggregation in time-based indicies in Elasticsearch.
- **invenio-queues**
  - Event queue management module.
- **counter-robots**
  - Module providing the list of robots according to the COUNTER Code of Practice.

1.2.6 Deposit bundle (alpha)

Note: This bundle is in alpha. The modules are being used in production systems but are very likely subject to change and are missing documentation.

Included modules:

- **invenio-deposit**
  - REST API for managing deposit of records into Invenio with support for in progress editing of records.
- **invenio-files-js**
  - AngularJS application for uploading files to Invenio via streaming the binary files in an HTTP request.
- **invenio-records-js**
  - AngularJS application for interacting with the deposit REST API and rendering forms based on angular schema forms.
- **invenio-sipstore**
  - Submission Information Package (SIP) store with bagit support.

1.2.7 Invenio modules (alpha)

Note: These modules are in alpha. The modules are being used in production systems but are most likely subject to changes and are missing documentation.

In addition to above bundles, we have a number of other individual modules which are all being used in production systems, but which are likely subject to change prior to final release and in most cases are missing documentation.

- **invenio-accounts-rest**
  - REST APIs for account management.
- **invenio-charts-js**
  - AngularJS application for producing charts.
- **invenio-csl-js**

  - AngularJS application for rendering citation strings via the records REST API and the CSL REST API.

- **invenio-csl-rest**
  - REST API for retrieving Citation Style Language (CSL) style files.
- **invenio-github**
  - GitHub integration with automatic archiving of new releases in Invenio.
- **invenio-openaire**
  - Integration with OpenAIRE, including support for harvesting Open Funder Regsitry and the OpenAIRE grants database, as well as REST APIs for funders and grants.
- **invenio-opendefinition**
  - REST API for licenses from OpenDefinition and SPDX.
- **invenio-pages**
  - Static pages module for Invenio.
- **invenio-pidrelations**
  - Persistent identifier relations management to support e.g. DOI versioning.
- **invenio-previewer-ispy**
  - ISPY previewer.
- **invenio-query-parser**
  - Invenio v1 compatible query parser for Invenio v3. Note the module is GPL licensed due to a GPL-licensed dependency.
- **invenio-records-editor**
  - JSON record editor.
- **invenio-records-editor-js**
  - Angular 4 application for editing JSON records.
- **invenio-s3**
  - Support for the S3 storage protocol in Invenio.
- **invenio-sequencegenerator**
  - Module for minting and tracking multiple sequences for e.g. report numbers, journals etc.
- **invenio-sse**
  - Server-Sent Events (SSE) integration in Invenio.
- **invenio-webhooks**
  - REST API for receiving and processing webhook calls from third-party services.
- **react-searchkit**
  - Modular React library for implementing search interfaces on top of Invenio, Elasticsearch or other search APIs. Replacement for Invenio-Search-JS.

1.2.8 Core libraries

Above Invenio modules dependent on a number of smaller core libraries we have developed to take care of e.g. identifier normalization, DataCite/Dublin Core metadata generation, testing and citation formatting.

- **citeproc-py-styles**
  - Citation Style Language (CSL) style files packaged as a Python module
- **datacite**
  - Python library for generating DataCite XML from Python dictionaries and registering DOIs with the DataCite DOI registration service.
- **dcxml**
  - Python library for generating Dublin Core XML from Python dictionaries.
- **dictdiffer**
  - Python library for diffing/patching/merging JSON documents.
- **dojson**
  - JSON to JSON rule-based transformation library.
- **flask-breadcrumbs**
  - Flask extension for managing breadcrumbs in web applications.
- **flask-celeryext**
  - Celery integration for Flask.
- **flask-iiif**
  - IIIF server for Flask.
- **flask-menu**
  - Menu generation support for Flask.
- **flask-sitemap**
  - Sitemaps XML generation for Flask.
- **flask-webpack**
  - Webpack integration for Flask.
- **idutils**
  - Persistent identifier validation, identification and normalization.
- **jsonresolver**
  - JSONRef resolver with support for local plugins.
- **pynpm**
  - NPM integration for Python.
- **pywebpack**
  - Webpack integration library for Python.
- **requirements-builder**
  - Python CLI tool for testing multiple versions of different Python libraries in you continuous integration system.

- **xrootdpyfs**
  - PyFilesystem plugin adding XRootD support.

1.2.9 Scaffolding

Following modules provide templates for getting started with Invenio:

- **cookiecutter-invenio-instance**
  - Template for new Invenio instances.
- **cookiecutter-invenio-datamodel**
  - Template for new data models.
- **cookiecutter-invenio-module**
  - Template for a reusable Invenio module.

1.2.10 Notes on license

Invenio is undergoing a change of license from GPLv2 to MIT License in most cases. Thus, you may especially for alpha and beta modules see that the license is still GPL v2 in the source code. This will be changed to MIT License for all repositories before being finally released. The only module we are currently aware of that can not be converted is Invenio-Query-Parser, which has a dependency on a GPL-licensed library. Invenio-Query-Parser is however not needed by most installations, as it only provides an Invenio v1.x compatible query parser.

1.3 Requirements

Following is a brief overview of the requirements of Invenio:

- **Supported Python versions:**
  - Python 3: v3.4, v3.5 and v3.6 (v3.7 is not yet supported due to Celery incompatibility).
  - Python 2: v2.7 (until 2020, the official end of life for Python 2.7)
- **Supported databases:**
  - PostgreSQL v9.4+, MySQL 5.6+ or SQLite (for testing).
- **Supported search engines:**
  - Elasticsearch v5 or v6.
- **Supported memory caches:**
  - Redis or Memcache.
- **Supported message queues:**
  - RabbitMQ, Redis or Amazon SQS (untested).
- **Supported storage protocols:**
  - Local, S3, WebDAV, XRootD and many more.
- **Supported WSGI servers:**
  - Gunicorn

  - uWSGI
  - mod_wsgi
- **Supported Flask versions:**
  - v0.12.x
  - v1.0.x
- **Supported Celery versions:**
  - v4.0, 4.1, 4.2
  - v3.1
- **Supported operating systems:**
  - macOS: Newer releases.
  - Linux: Newer distributions such as Ubuntu 16/18 or CentOS 7.

1.3.1 System requirements

Invenio can run in both Docker, virtual machines and physical machines. Invenio can run on a single machine or a cluster of 100s of machines. It all depends on exactly how much data you are handling and your performance requirements.

Following is an estimate of a small, medium and large installation of Invenio. The purpose is only to provide a **very rough** idea about how an Invenio installation could look like.

Small installation:

- Web/app/background servers and Redis: 1 node
- Database: 1 node
- Elasticsearch: 1 node

Medium installation:

- Load balancer: 1 node
- Web/app servers and background workers: 2 nodes
- Database: 1 node
- Elasticsearch: 3 nodes
- Redis/RabbitMQ: 1 node

Large installation:

- Load balancer: 2 node (with DNS load balancing)
- Web/app servers: 3+ nodes
- Background workers: 3+ nodes
- Database: 2 nodes (master/slave)
- Elasticsearch: 5 nodes (3 data, 2 clients)
- Redis: 3 nodes (HA setup)
- RabbitMQ: 2 nodes (HA setup)

1.4 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 exsiting 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.

CHAPTER 2

Quickstart

2.1 Launch an Invenio instance

2.1.1 Prerequisites

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

- Docker v1.18+ and Docker Compose v1.23+
- NodeJS v6.x+ and NPM v4.x+
- Enough virtual memory for Elasticsearch (when running in Docker).
- Cookiecutter
- Pipenv

Invenio uses Cookiecutter to scaffold the boilerplate for your new instance and uses Pipenv to manage Python dependencies in a virtual environment. Above links contain detailed installation instructions, but the impatient can use following commands:

```
# Install cookiecutter if it is not already installed
$ sudo apt-get install cookiecutter
$ sudo apt-get install pipenv
# or e.g.
$ pip install --upgrade cookiecutter pipenv
```

2.1.2 Scaffold

First step is to scaffold a new instance using the official Invenio cookiecutter template.

```
$ cookiecutter gh:inveniosoftware/cookiecutter-invenio-instance --checkout v3.1
# ...fill in the fields...
```

Note, the cookiecutter script will ask you to resolve some TODOs. These will be covered in the *Next Steps* section of this quick start guide.

The scaffolded instance comes by default with a toy example data model to help you get started.

2.1.3 Install

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:

First, fire up the database, Elasticsearch, Redis and RabbitMQ:

```
$ cd my-site/
$ docker-compose up -d
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.

Next, activate the virtualenv of the new project by running:

```
$ pipenv shell
```

Finally, install all dependencies, build the JS/CSS assets, create the database tables and create the Elasticsearch indices by running the bootstrap and setup scripts:

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

2.1.4 Run

You can now start the development web server and the background worker for your new Invenio instance:

```
(my-site)$ ./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/ !

Continue tutorial

Create, Display, Search Records

2.2 Create, Display, Search Records

2.2.1 Create a record

By default, the toy 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.

2.2.2 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,
```

(continues on next page)

(continued from previous page)

```
  "updated": "2018-05-23T13:28:19.426213+00:00"
}
```

2.2.3 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": {
    "keywords": {
      "buckets": [],
      "doc_count_error_upper_bound": 0,
      "sum_other_doc_count": 0
    },
    "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,
          "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"
  }
}
```

Continue tutorial

Next Steps

2.3 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 the 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. 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 in your site directory:

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

2. 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:

   ```
   # if you have activated the virtual environment skip `pipenv shell`
   $ pipenv shell
   (my-site)$ python setup.py extract_messages
   (my-site)$ python setup.py init_catalog -l en
   (my-site)$ python setup.py compile_catalog
   ```

   Transifex

   Make sure you edit `.tx/config` and sign-up for Transifex before trying below steps.

   Install the transifex-client

   ```
   (my-site)$ pipenv install transifex-client
   ```

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

   ```
   (my-site)$ tx push --skip --translations
   ```

   Pull translations for a single language from Transifex

   ```
   (my-site)$ tx pull --language en
   ```

3. REST API permissions: By default your Invenio instance have no permissions enabled, which means that any user will be allowed to perform any operation (read, update, create and delete) over the records. Check *Managing access to records* for information on how to adapt the permissions to your needs.

2.3.1 Testing

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

```
# run all the tests...
(my-site)$ ./run-tests.sh
# ...or to run individual tests
(my-site)$ pytest tests/test_version.py
```

2.3.2 Documentation

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

```
(my-site)$ python setup.py build_sphinx
```

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

CHAPTER 3

Tutorials

We maintain tutorials and training material for Invenio v3 in a separate repository:

- https://github.com/inveniosoftware/training

3.1 Prerequisites

- Prerequisites

3.2 Invenio at a glance

- Tutorial 01 - Getting started
- Tutorial 02 - Tour of Invenio
- Tutorial 03 - Infrastructure overview

3.3 Starting development with Invenio

- Tutorial 04 - Running Invenio
- Tutorial 05 - Customizing the look and feel
- Tutorial 06 - Developing with Invenio

3.4 Learning about data models

- Tutorial 07 - Adding a new field

- Tutorial 08 - From scratch
- Tutorial 09 - Build a simple deposit form
- Tutorial 10 - Modify records before indexing
- Tutorial 11 - Linking records
- Tutorial 12 - Managing access

3.5 Invenio in production

- Tutorial 13 - Securing your instance
- Tutorial 14 - Deployment and monitoring

3.6 Advanced topics

- Tutorial 15 - Application architecture

CHAPTER 4

Build a repository

Now that you have generated the skeleton of your repository via cookiecutter, bootstrapped your instance, run it and created a record, let's have a deep dive at the various files and folders that were generated and what they do.

The generated skeleton is our default recommendation, however you are completely free to adapt it as your see fit.

4.1 Management scripts

```
...
├── scripts
│   ├── bootstrap
│   ├── console
│   ├── server
│   ├── setup
│   └── update
...
```

In your root folder, you will find the `scripts` directory which contains executable bash scripts that will assist you with developing and managing your Invenio instance:

scripts/bootstrap

Installs all of the Python dependencies, your application's code, and collects and builds the static files required for the instance to run. We'll talk more about how we manage *dependencies* in the relevant section below.

scripts/setup

(Re)initializes data needed for services that hold application state, i.e.:

- Database tables
- Elasticsearch indices and templates
- RabbitMQ queues

- Redis databases

This script is also useful when you're doing local development and want to start from a clean state.

Warning: This scripts performs destructive and non-reversible operations. Only run this when you initialize your instance the first time. Running this in e.g. a production or testing environment will remove all existing data.

scripts/server

Fires up a development HTTPS-enabled flask web server at https://localhost for your application and a Celery worker. As you make HTTP requests to the web application or run any tasks you will see information, warnings and errors being logged in the terminal. Interrupting this script will automatically stop both services.

scripts/console

This will spawn an interactive IPython shell with your application fully loaded. You can use it to run arbitrary Python commands while having access to your application's database models for queries. This is a great tool for testing functionality during development, troubleshooting and fixing problems on a live instance

scripts/update

This will repeat all of the steps of the `bootstrap` script, but will also additionally apply any new Alembic recipes for the database and Elasticsearch index changes.

4.2 Python dependencies and packaging

```
...
├── Pipfile
├── Pipfile.lock
├── setup.py
├── MANIFEST.in
...
```

To manage our Python dependencies we have chosen to use pipenv. Pipenv does the following:

- Tracks your *loose* Python dependencies inside `Pipfile`.
- Pins specific versions (and hashes) of your Python depedendencies inside `Pipfile.lock`. The existence of this file is essential to make sure that when you deploy your instance on a production environment, you can reproduce the exact same environment that you used when you developed and tested your application.
- Automatically creates a Python virtualenv with the correct Python version under the path defined in the `WORKON_HOME` environment variable (commonly used by `virtualenvwrapper`). If not set, new virtualenvs will be placed under `$HOME/.local/share/virtualenvs/`.

We still need a `setup.py` file though, not for tracking any dependencies, but for specifiyng the entrypoints that various Invenio packages rely on to automatically detect and register Flask blueprints, Celery tasks and other features.

4.3 Docker and Docker-Compose

```
...
├── docker
│   ├── postgres
│   │   ├── ...
│   ├── uwsgi
│   │   ├── ...
│   ├── nginx
│   │   ├── ...
│   ├── haproxy
│   │   ├── ...
├── docker-services.yml
├── docker-compose.yml
├── docker-compose.full.yml
├── Dockerfile.base
├── Dockerfile
...
```

The instance requires some services in order to run, like a database, Elasticsearch, Redis and RabbitMQ. To provide a cross-platform and convenient way of running these services, we are using Docker and Docker Compose, by configuring the following files:

docker-services.yml

This file contains basic definitions for the Docker containers for the services the instance uses. Configuration options such as the database credentials, exposed ports, and other service-specific options can be modified in here. This file's containers are used as a common base and are extended by other `docker-compose.*.yml` files to build up a specific configuration for an infrastructure.

docker-compose.yml

This file contains and exposes locally the minimal set of service containers needed for developing the instance locally:

- `db`: The database, PostgreSQL or MySQL, exposing the 5432 or 3306 ports.
- `es`: Elasticsearch version 5 or 6, exposing the 9200 and 9300 ports.
- `mq`: RabbitMQ, exposing port 5672 for the service and port 15672 for a management web server (accessible via the default username/password `guest:guest`).
- `cache`: Redis exposing port 6379.

When developing and running your instance locally these services can be accessed by your application.

docker-compose.full.yml

This file contains a full-fledged definition of a production-like application infrastructure. It has all of the `docker-compose.yml` file's containers defined, and additionally:

- `lb`: HAProxy, publicly exposing ports 80 and 443 for accessing the web application and 8080 for accessing statistics.
- `frontend`: Nginx, exposing ports 80 and 443 and acting as a reverse proxy for your application containers and serving static files.
- `web-ui`/`web-api`: Two separate web application containers running uWSGI for the Invenio UI and REST API applications and exposing port 5000
- `worker`: The Celery worker of your application.

- `flower`: Monitoring web application for Celery, publicly exposing port 5555.
- `kibana`: Monitoring web application for Elasticsearch, publicly exposing port 5601.

The `web-ui`, `web-api` and `worker` containers are using Docker images that are built from the `Dockerfile.base` and `Dockerfile` files described below.

> **Warning:** While one might be tempted to deploy this as a fully functional Invenio instance, it is not meant to be a turn-key solution, since it hasn't been tested for this purpose. This is rather meant to be an inspiration (in terms of configuration, networking and general principles) for configuring your own setup either by replacing the container services with actual nodes/machines or configuring a production-level container orchestration system like Kubernetes, OpenShift, etc.

Dockerfile.base

This Dockerfile helps you build a Python dependencies-only base image from where your application can be built quickly.

Dockerfile

This Dockerfile builds a fully functional image of your application with all of the static assets it requires.

docker/postgres

Contains a Dockerfile and script that will setup the necessary users/roles for the database.

docker/uwsgi

Contains a the `uwsgi_ui.ini` and `uwsgi_api.ini` uWSGI configruation files used for running the Invenio UI and REST API web applications.

docker/nginx

Contains a Dockerfile, nginx configurations (`nginx.conf` and `conf.d/default.conf`) and a self-signed generated SSL certificate (`test.crt` and `test.key`). You can look into these files if you are interested in how to confiugre nginx to proxy requests to one or multiple uWSGI web application.

docker/haproxy

Contains a Dockerfile, HAProxy configuration (`haproxy.cfg`) and a self-signed generated SSL certificate (`haproxy_cert.pem`).

4.4 Configuration

```
...
├── my_site
│   ├── config.py
...
```

my_site/config.py

The instance's basic configuration variables are defined inside this file. You should go through all of these variables to understand what kind of things can be customized for your instance, like e.g. what should be the "From" email address for your automatically sent emails.

The configuration used by the Invenio applications is dynamically loaded from multiple sources. You can read more about this in Invenio-Config documentation. Probably the most important part of this, is the order in which the various configuration sources are loaded, which allows you to effectively override any config variable. The following list describes this order (every item overrides the one above it):

- Configuation modules defined in `invenio_config.module` entrypoints. `my_site.config` is actually one of them. You can add as many as you want and they will be applied in alphabetical order of the entrypoint name.
- Configuration in the `<app.instance_path>/invenio.cfg`. For local development this is usually `${VIRUAL_ENV}/var/instance/invenio.cfg`.
- `INVENIO_XYZ` environment variables. If for example you want to override the `SECRET_KEY`, you would have to do `export INVENIO_SECRET_KEY="my-secret"`.

4.5 Tests

```
...
├── tests
│   ├── api
│   │   ├── conftest.py
│   │   └── test_api_simple_flow.py
│   ├── e2e
│   │   ├── conftest.py
│   │   └── test_front_page.py
│   ├── ui
│   │   └── conftest.py
│   ├── conftest.py
│   └── test_version.py
├── pytest.ini
├── run-tests.sh
...
```

In Invenio we're using the Python pytest library for testing. All of the instance's tests are placed in the `tests/` directory.

tests/ui/

Includes tests that use the UI application views.

tests/api/

Includes tests that use the REST API application views.

tests/e2e/

Includes Selenium-based end-to-end tests which access both the UI and REST API applications.

pytest.ini

Used to configure `pytest` and its various plugins.

run-tests.sh

You can run this script locally or in your CI/CD pipeline and it will check:

- Your Python dependencies for security vulnerabilities using pyup.io's "safety" library.
- Your docs styling based on PEP 257.
- Your Python import for the correct sorting order using isort.
- Your `MANIFEST.in` for any missing entries.
- Your docs are building without errors.
- That your tests are passing.

4.6 Documentation

```
...
├── docs
│   ├── api.rst
│   ├── authors.rst
│   ├── changes.rst
│   ├── configuration.rst
│   ├── conf.py
│   ├── contributing.rst
│   ├── index.rst
│   ├── installation.rst
│   ├── license.rst
│   ├── make.bat
│   ├── Makefile
│   ├── requirements.txt
│   └── usage.rst
├── AUTHORS.rst
├── CHANGES.rst
├── CONTRIBUTING.rst
├── INSTALL.rst
├── README.rst
...
```

To build the instance's documentation we're using Sphinx docs and reStructuredText as a markup language.

docs/*.rst

The various `.rst` files are placed in the root of your repository and in the `docs/` directory, and will be used to build your instance's documentation, via running `pipenv run build_sphinx`.

docs/conf.py

This is the place where various documentation configuration variables can be set. You can have a look at it and tweak things based *Sphinx docs' extensive section on its configuration <http://www.sphinx-doc.org/en/master/usage/configuration.html>*

CHAPTER 5

Understanding data models

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.

5.1 First steps

First of all, make sure you have followed the *Quickstart*, 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_site
|   |-- config.py
|   |-- records
|   |   |-- jsonschemas/
```

(continues on next page)

(continued from previous page)

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

Steps

Building a data model involves the following tasks:

- **Internal representation**
  - *Define a JSONSchema* – used to validate the internal structure of your record.
  - *Define an Elasticsearch mapping* – used to specify how your records are indexed by the search engine.
- **External representation**
  - *Define serializers* – transform an *internal* representation to an external (e.g. JSON to DataCite XML).
  - *Define loaders* – transform and validates an *external* representation to an internal (e.g. DataCite XML to JSON).
  - *Define a Marshmallow schema* – used to build loaders and serializers.
- **Exposing records via the UI and REST API**
  - *Define templates* – used to render search results and landing pages.
  - *Configure the UI* – enables HTML landing pages for your records.
  - *Configure the REST API* – enables the REST API for your records.

5.2 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_site
|   |-- records
|   |   |-- 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",
```

(continues on next page)

(continued from previous page)

```
    "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.

5.3 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_site
|   |-- records
|   |   |-- mappings
|   |   |   |-- __init__.py
|   |   |   |-- v5
|   |   |   |   |-- __init__.py
|   |   |   |   `-- records
```

(continues on next page)

(continued from previous page)

```
|   |   |   |       `-- 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,
            "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, therefore, this field is searched based on exact match. 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.

5.3.1 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_site
|   |-- records
|   |   |-- 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",
    "...": "..."
}
```

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_site
|    |-- records
|    |    |-- 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'
        ],
        # ...
```

(continues on next page)

(continued from previous page)

```
    },
)
```

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.

5.4 Define a Marshmallow schema

Marhsmallow 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_site
|   |-- records
|   |   |-- 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 deserialization (think of it as form validation).

5.5 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_site
|   |-- records
|   |   |-- 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*.

5.6 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_site
|    |-- records
|    |    |-- 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.

5.7 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_site
|    |-- records
|    |    |-- 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.

5.8 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 data model's `config.py`:

```
|-- my_site
|    |-- records
|    |    |-- config.py
```

Note: Take care, not to confuse `my_site/records/config.py` (the data model's module configuration) with `my_site/config.py` (your application's configuration).

To avoid the application configuration file from growing very big, we usually keep the **default** configuration for a module in a `config.py` inside the module.

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`).

5.9 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>',
    ),
}
```

5.10 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

CHAPTER 6

Managing access to records

Invenio comes with a access control system that is very powerful and flexible but which can also seem overwhelming at first. This guide will show you a basic example of how to protect our REST API so only some users can see certain records.

If you haven't already done so, make sure you've followed the *Quickstart* so you have an Invenio instance to work on.

6.1 Endpoints

Your data model's REST API has two main endpoints:

- Search endpoint (e.g. `/api/records`)
- Detail endpoint (e.g. `/api/records/<id>`)

The goal is to ensure that a) only an owner of a record can retrieve their record from the detail endpoint and b) that the search endpoint only shows records that a given user owns.

In order to protect the two endpoints we will:

1. **Store permissions** in a record by adding a new field `owner` to our data model.
2. **Require permissions by writing:**
    - a *permission factory* to protect the detail endpoint.
    - a *search filter* to filter results in the search endpoint.
3. **Configure endpoints** to use the permission factory and search filter.

6.2 Storing permissions

First, below is an example of how we could store an owner inside a record by adding a new field `owner`:

```
{
    "title": "My secret publication",
    "owner": 1
}
```

In order to be able to store the `owner` property in our data model, you must add this new field to the data model's:

1. **JSONSchema** (think of the JSONSchema as the database table structure that you add a new column to).
2. **Elasticsearch mapping** (think of the mapping as a description of how your data should be indexed).
3. **Marshmallow schema** (think of the Marshmallow schema as a description of how you would render one or more rows in a database table to an end-user).

6.2.1 JSONSchema

In the quickstart example, our JSONSchema is located in `.../records/jsonschemas/record-v.1.0.0.json`, and you would add something like below to our schema:

```
{
    "owners": {
    "type": "integer"
    }
}
```

6.2.2 Elasticsearch mapping

In the quickstart example, our Elastisearch mapping is likely located in `.../records/mappings/v6/record-v.1.0.0.json`, and you would add something like below to our mapping:

```
{
    "owners": {
    "type": "int"
    }
}
```

6.3 Requiring permissions

Once you have added the new field(s) to your data model, you need to make use of the field to protect the detail endpoint and the search endpoint. You do that by writing

- a *permission factory* to protect the detail endpoint.
- a *search filter* to filter results in the search endpoint.

6.3.1 Permission factory

The purpose of the permission factory is to create a permission object from a record which is then used by the detail endpoint to check if the current user has permission to view the current record. Below is a simple example of a permission factory:

```
from invenio_access import Permission
from flask_principal import UserNeed

def my_permission_factory(record=None):
    return Permission(UserNeed(record["owner"]))
```

The permission factory function takes as input a record and creates a `Permission` object from it.

The permission, when checked, requires that the current user has the same id as the id stored in the records `owner` field. This is expressed with the `UserNeed`.

Permissions and needs

The concept of *needs* can be somewhat hard to grasp, but essentially it just expresses the smallest level of access control. For instance `UserNeed(1)` expresses the statement "has user id 1", and `RoleNeed('admin')` expresses the statement "has admin role".

A *permission* represents a set of required *needs*. For instance `Permission(UserNeed(1), RoleNeed('admin'))` expresses the statement "has user id 1 or has admin role".

Thus, with a permission factory you can build arbitrarily complex permissions from the information stored in your records.

6.3.2 Search filter

For searches over possibly millions of records we need to be able to efficiently check permissions of all records. This is done with a search filter which is applied when executing a query. In comparison, a permission factory only deals with one record at a time.

Below is an example of search filter which is applied to all queries on the search endpoint:

```
from elasticsearch_dsl import Q
from flask_security import current_user
from invenio_search.api import DefaultFilter, RecordsSearch

def permission_filter():
    return [Q('match', owner=current_user.get_id())]

class MyRecordSearch(RecordsSearch):
    class Meta:
        index = 'records'
        default_filter = DefaultFilter(permission_filter)
```

The method `permission_filter` when called, will create an Elasticsearch DSL `Q()` (query object) which will match all records where the property owner equals the current user's id (`current_user` is an object that holds the current request's authenticated user).

The class `MyRecordSearch`, will be responsible for executing all queries on the search endpoint. In above example, we set the name of the Elasticsearch index it should used, and the search filter which it should use (in our case the permission filter).

Search filter vs permission factory

There's a subtle difference between the search filter and the permission factory which is worth noting.

The permission factory takes a record as input, while the search filter takes the current user as input. For the permission factory, the created permission is checked against the current user, while with the search filter the current user is checked against the records. Hence, the permission factory and search filter are coming from each their end when checking permissions.

It's therefore very important when writing the search filter and permission factory, that the two are producing identical results.

6.4 Configuring endpoints

The last part of the puzzle is to tell our detail/search endpoints to use our newly created permission factory and search filter:

```
RECORDS_REST_ENDPOINTS = {
    'recid': dict(
        # ...
        search_class=MyRecordSearch,
        read_permission_factory_imp=my_permission_factory,
        # ...
    ),
}
```

In our case we are protecting only the read operation on the view. Needless to say, as the REST API also supports CRUD operations, you should also protect the other operations with their a permission factory.

6.5 Complex access rights

The toy example presented in this guide is too simple for most normal requirements, thus in order to provide some inspiration, we here present two more complex ways you could store access rights in records:

6.5.1 Computed rights

In some cases, it can be an advantage to use existing properties in your record to manage access rights. This way, you ensure that access rights does not get out of sync with other properties. An example of such a record could be:

```
{
    "visibility": "restricted",
    "owners": [1, 2],
    "communities": ["blr"]
}
```

A permission factory could for above record then compute different permissions objects for different types of actions.

For reading the record, the permission could be:

```
Permission(any_user)
```

For seeing the files in the record, the permission could be:

```
Permission(UserNeed(1), UserNeed(2), RoleNeed('blr-curators'))
```

For editing the record, the permission could be:

```
Permission(UserNeed(1), UserNeed(2))
```

6.5.2 Explicit rights

In some cases, it is an advantage to have explicit rights defined on your record so that even if the code changes, it still obvious who should have access for which actions. An example of such a record could be:

```
{
    "_access": {
        "read": {
            "systemroles": ["campus_user"]
        },
        "update": {
            "users": [1],
            "roles": ["curators"],
        }
    }
}
```

This way, changes to rights can also be explicitly tracked via the records revision history and thus be audited.

6.5.3 Further information

- Invenio-Access
- Invenio-Records-REST

CHAPTER 7

Securing your instance

Invenio has a core principle to always have secure defaults for settings, thus we have already done a lot in order to secure your installation. It is however still important to be aware of some important settings and risks.

7.1 Secret key

Probably the most important security measure is to have a strong random secret key set for you Invenio instance. The secret key is used for instance to sign user session ids and encrypt certain database fields.

The secret key must be kept secret. If the key is leaked or stolen somehow, you should immediately change it to a new key.

```
# config.py
SECRET_KEY = '..put a long random value here..'
```

Good practices:

- Never commit your secret key in the source code repository (or any other password for that sake).
- Use different secret keys for different deployments (testing, staging, production)

7.2 Allowed hosts

Invenio has a configuration option called `APP_ALLOWED_HOSTS` which controls which hosts/domain names that can be served. A client request to a web server usually includes the domain name in the `Host` HTTP header:

```
GET /
Host: example.org
...
```

The web server uses that for instance to host several website on the same domain. Also, the host header is usually used in a load balanced environment to generate links with the right domain name.

An attacker has full control of the host header and can thus change it to whatever they like, and for instance have the application generate links to a completely different domain.

Normally your load balancer/web server should only route requests with a white-listed set of host to your application. It is however very easy to misconfigure this in your web server, and thus Invenio includes a protective measure.

Simply set `APP_ALLOWED_HOSTS` to a list of allowed hosts/domain names:

```
# config.py
APP_ALLOWED_HOSTS = ['www.example.org']
```

7.3 Number of proxies

Invenio is commonly used with both a load balancer and a web server in front of the application server. The load balancer and web server both works as proxies, which means that the clients remote address usually get's added in the `X-Forwarded-For` HTTP header. Invenio will automatically extract the clients IP address from the HTTP header, however to prevent clients from doing IP spoofing you need to specify exactly how many proxies you have in front of you application server:

```
# config.py
WSGI_PROXIES = 2
```

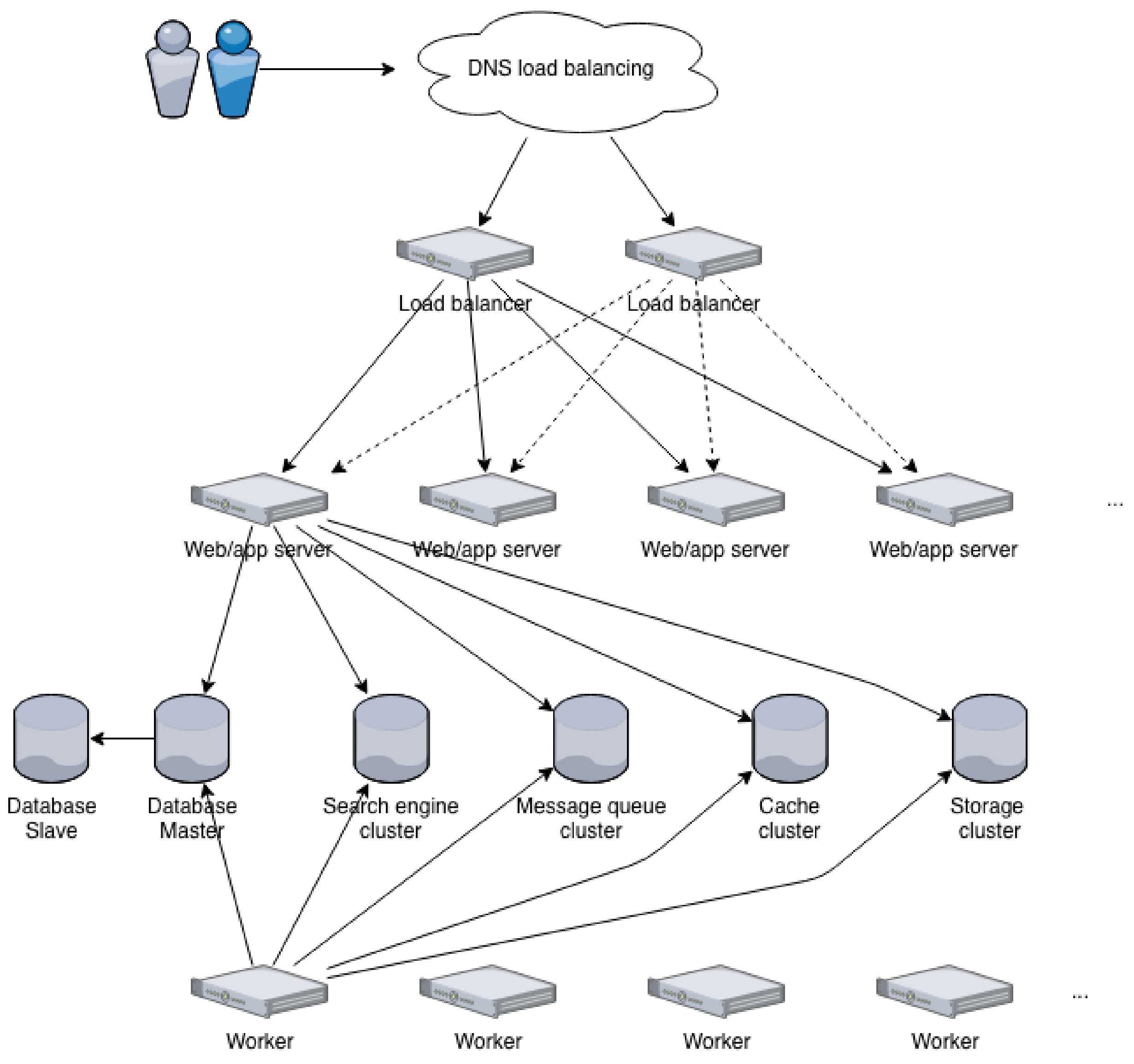
CHAPTER 8

Infrastructure architecture

This guide provides a general overview of the Invenio infrastructure architecture. It is not meant to be a comprehensive guide for each subsystem.

Over all, the Invenio infrastructure is a pretty standard web application infrastructure. It consists of:

- **Load balancers:** HAProxy, Nginx or others.
- **Web servers:** Nginx, Apache or others.
- **Application servers:** UWSGI, Gunicorn or mod_wsgi.
- **Distributed task queue:** Celery
- **Database:** PostgreSQL, MySQL or SQLite.
- **Search engine:** Elasticsearch (v5 and v6).
- **Message queue:** RabbitMQ, Redis or Amazon SQS.
- **Cache system:** Redis or Memcache.
- **Storage system:** Local, S3, XRootD, WebDAV and more.



8.1 Request handling

A client making a request to Invenio will usually first hit a load balancer. For high availability you can have more load balancers and balance traffic between them with e.g. DNS load balancing.

8.1.1 Load balancer

Request types

The load balancer usually (if it supports SSL termination) allows you to split traffic into three categories of requests:

- static files requests: e.g. javascript assets
- application requests: e.g. search queries
- record files requests: e.g. downloading very large files

This way you can dimension the number connection slots between different types of requests according to available resources. For instance a static file request can usually be served extremely efficiently, while an application request usually takes longer and requires more memory.

Similar, downloading a very file depends on the client's available bandwidth and can thus take up a connection slot for a significant amount time. If your storage system supports it, it is possible with Invenio to completely offload the serving of large files to your storage system (e.g. S3).

All in all, the primary job of the load balancer is to manage traffic to your servers according to available resources. For instance during traffic floods the load balancer takes care of queue requests to the web servers.

Backup pages

A load balancer can also direct traffic to a static backup site in case your main web server is down. This is useful in order to communicate with users during major incidents.

8.1.2 Web servers

The load balancer proxies traffic to one of several web servers. The web server's primary job is to manage the connections into your application server. A web server like Apache and Nginx is usually much better than an application server to manage connections. Also, you can use the web server to configure limits on specific parts of your application so that for instance you can upload a 1TB file on the Files REST API, but not on the search REST API.

8.1.3 Application servers

The web server proxies traffic usually (but not necessarily) to a single application server running on the same machine. The application server is responsible for handling the application requests. Invenio is a Python application, and thus make use of the WSGI standard. There exists several application servers capable of running WSGI python application, e.g. Gunicorn, uWSGI and mod_wsgi.

8.2 Storing records

Invenio store records as JSON documents in an SQL database. Most modern SQL databases today have a JSON type, that can efficiently store JSON documents in a binary format.

Transactional databases

The primary reason using an SQL database is that they provide transactions, which is very important since data consistency for a repository is of utmost importance. Also, database servers can handle very large amounts of data as long as they are scaled and configured properly. Last but not least, they are usually highly reliable as compared to some NoSQL solutions.

Primary key lookups

Most access from Invenio to the database is via primary key look ups, which are usually very efficient in database. Search queries and the like are all sent to the search engine cluster which can provide much better performance than a database.

8.3 Search and indexing

Invenio uses Elasticsearch as its underlying search engine since Elasticsearch is fully JSON-based, and thus fit well together with storing records internally in the database as JSON documents.

Elasticsearch furthermore is highly scalable and provides very powerful search and aggregation capabilities. You can for instance make geospatial queries with Elasticsearch.

8.3.1 Direct indexing

Invenio has the option to directly index a record in Elasticsearch when handling a request, and thus make the record immediately available for searches.

Bulk indexing

In addition to direct indexing, Invenio can also do bulk indexing which is significantly more efficient when indexing large number of records. The bulk indexing works by the application sending a message to the message queue, and at regular intervals a background job will consume the queue and index the records. Also, several bulk indexing jobs can run concurrently at the same time on multiple worker nodes and thus you can achieve very high indexing rates during bulk indexing.

8.4 Background processing

Invenio relies on an application called Celery for distributed background processing. In order for an application server to reply faster to a request, it can offload some task to asynchronous jobs. It works by the application sending a message to the message queue (e.g. RabbitMQ), which several Celery worker nodes continuously consume tasks from.

An example of background tasks can for instance be sending an email or registering a DOI.

Multiple queues

The background processing supports multiple queues and advanced workflows. You could for instance have a low priority queue that constantly runs x number of file integrity checks per day, and another normal queue for other tasks like DOI registration.

Cronjobs and retries

Celery also supports running jobs at scheduled intervals as well as retrying tasks in case the fail (e.g. if a remote service is temporarily down).

8.5 Caching and temporary storage

Invenio uses an in-memory cache like Redis or Memcache for fast temporary storage. The cache is for instance used for:

- User session storage
- Results from background jobs
- Caching rendered pages

8.6 Storing files

Invenio comes with a default object storage REST API to expose files. Underneath the hood, Invenio can however store files in multiple different storage systems due to a simple storage abstraction layer. Also, it is possible to completely by-pass the Invenio object storage and directly use another storage system like S3. In this case, you just have to be careful to manage access correctly on the external system.

Multiple storage systems

One strength of Invenio is that you can store files on multiple systems at the same time. This is useful if you for instance need to use multiple systems or do live migration from one system to another.

CHAPTER 9

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 web server, 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 make up Invenio, and which also allow you to easily extend Invenio with your own modules.

9.1 Core concepts

We will explain the core Flask concepts using a 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 with just 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 reuse the blueprint in another Flask application.

Flask extensions

Like blueprints allow you to modularise your Flask application's views, Flask extensions allow you to modularise the initialization of your application that is not specific to views (e.g. providing database connectivity).

Flask extensions are just objects like the one in the example above, which has an `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 the above example, 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.

9.2 Interfaces: WSGI, CLI and Celery

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

- **WSGI:** The frontend web server 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.

9.3 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__, 'bp2')
@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.

9.4 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. This essentially sets all the configuration variables for which 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 provide 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 the 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, within 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 with, so e.g. Flask extensions are loaded before any blueprints are loaded.

9.5 Module discovery

In each of the application assembly phases, the Invenio factory automatically discovers your installed Invenio modules. This works 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. it provides.

9.6 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 on 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 returns a JSON response in the REST API application.

9.7 Implementation

The following Invenio modules are each responsible for implementing parts of the above application architecture, and it is highly advisable to dig deeper into these modules for 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.

CHAPTER 10

Migrating to v3

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.

10.1 Dumping data from v1.2

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

10.1.1 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.

10.1.2 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
```

10.1.3 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.

10.2 Loading data in v3

10.2.1 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
```

10.2.2 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.

10.2.3 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 11

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).

11.1 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.

11.1.1 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/
```

11.2 Configuring Invenio

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

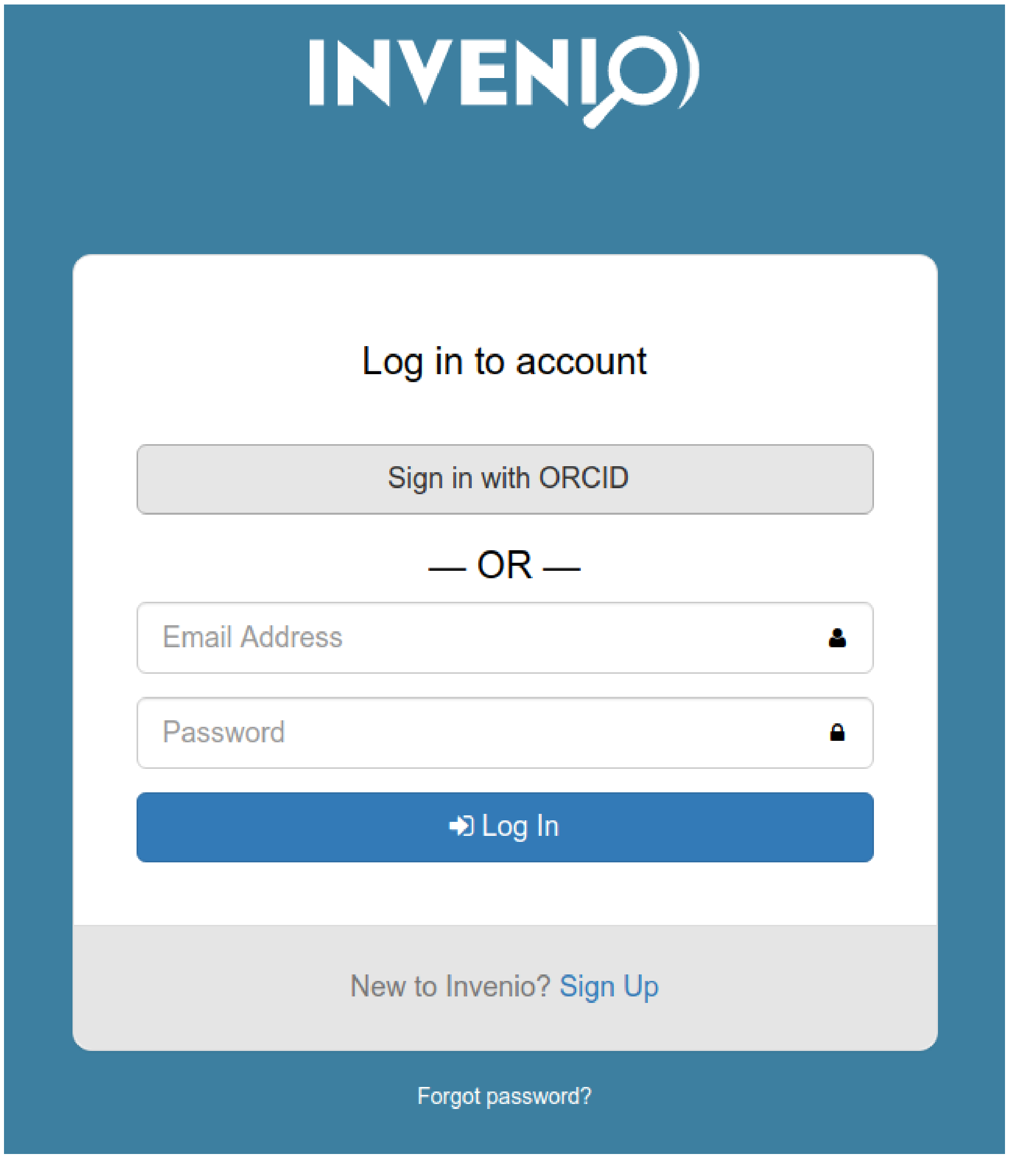
```
from invenio_oauthclient.contrib import orcid

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:



11.2.1 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.

CHAPTER 12

History

12.1 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.

12.2 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.

CHAPTER 13

Releases

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

13.1 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.

13.1.1 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.

13.1.2 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.

13.1.3 End of life dates

Release	Earliest EOL Date	Maintained until
v3.1.x	2020-03-31	v3.3.0
v3.0.x	2019-06-07	v3.2.0
v2.x.y	2018-06-07	v3.0.0
v1.x.y	2018-06-07	v3.0.0

13.2 Version 3.1.2

Released 2020-02-11

Invenio v3.1.2 fixes issues with an incompatibility of the recent released Werkzeug v1.0

13.2.1 Maintenance policy

Invenio v3.1 will be supported with bug and security fixes until the release of Invenio v3.3 and minimum until 2020-03-31.

See our *Maintenance Policy*.

13.3 Version 3.1.1

Released 2019-07-15

Invenio v3.1.1 fixes two security issues.

13.3.1 Security fixes

- **Invenio-App:** Fixed a Host header injection vulnerability.
- **Invenio-Records:** Fixed a Cross-Site Scripting vulnerability in the administration interface.

13.3.2 Maintenance policy

Invenio v3.1 will be supported with bug and security fixes until the release of Invenio v3.3 and minimum until 2020-03-31.

See our *Maintenance Policy*.

13.4 Version 3.1.0

Released 2019-03-11

We are proud to announce the release of Invenio v3.1.0.

Python compatibility

Invenio v3.1 supports Python 2.7 (until 2019-12-31), Python 3.5 and Python 3.6. We expect to add support for Python 3.7 in the near-term future once Celery v4.3 has been released.

13.4.1 Getting started

See our *Quickstart* guide.

13.4.2 What's new in Invenio v3.1?

Webpack build system

Invenio v3.1 comes with a new assets build system based on Webpack for building and packaging your JavaScript applications, stylesheets and much more. The system replaces the previous AMD/RequireJS based system which was deprecated in v3.0.

The old build system is still available to allow users to upgrade to Invenio v3.1 without first migrating to Webpack. The old build system will be removed in Invenio v3.3

For more information about the new build system, please see:

- Invenio-Assets
- Upgrade to Webpack

The new Webpack build system is based on:

- Flask-WebpackExt
- Pywebpack

Simplified scaffolding

We have simplified the scaffolding of new Invenio instances by merging the data model template into the main Cookiecutter-Invenio-Instance.

You can try the new approach by following our *Quickstart*.

The previous approach of two separate packages – one for the application and one for the data model – caused friction and confusion for new users and we therefore decided to merge both.

This also fits with our *long-term* goal, where we want to provide standard data models (such as DataCite, Dublin Core, MARC21) so that users don't have write their own data model.

Docker base image

We have released a new Docker image that can serve as a base image for your Invenio instances. The image is based on CentOS 7 and comes with Python 3.6, Node.JS, NPM and some standard libraries often needed by Invenio.

See inveniosoftware/centos7-python on DockerHub.

The new image is being used by our *Quickstart* guide, and the image is usable in production environments like OpenShift.

Pipenv

In order to manage Python dependencies more reliable and securely for your Invenio instance we have moved to use Pipenv which also handles the virtualenv creation.

Pipenv has been integrated into the scaffolded Invenio instance, and you can read more about it in *Build a repository*.

Documentation

New sections where added to the documentation specifically on:

- *Bundles* and *Requirements*
- *Build a repository*
- *Managing access to records*
- *Securing your instance*
- *Infrastructure architecture*

Request tracing

Invenio v3.1 has added new features for improved request tracing to allow for better troubleshooting and auditing of problems. The feature allows logging a request id, session id and user id across multiple services such as Nginx and Invenio error logs. This enables e.g. system administrators to identify exactly which Nginx access log line caused a specific error logged by Invenio.

If combined with e.g. centralised log aggregation, this can be used for e.g. viewing requests by a user in real-time, request performance statistics and many other metrics. Please note that in order to be compliant with EU General Data Protection Regulation (GDPR), you must ensure that these logs are automatically deleted after 3 months (the same is the case if you only log an IP address).

- **Cookiecutter-Invenio-Instance:**
  - Nginx configuration has been updated to automatically generate a random request id and add is as `X-Request-ID` header.
  - Nginx log format has been updated to log timing information, request id, session id and user id if provided by the application server in the `X-Session-ID` and `X-User-ID` HTTP headers. Nginx will remove both headers prior to sending the response to the client.
- **Invenio-App:**
  - Extracts the `X-Request-ID` header (max 200 chars) if set in the HTTP request and makes it available on the Flask `g` object as `g.request_id`.
- **Invenio-Logging:**
  - The request id is made available to all log handlers.
  - The Sentry log handler will add the request ID as a tag if available.
- **Invenio-Accounts**
  - The `X-Session-ID` and `X-User-ID` HTTP headers will be added to the HTTP repsponse if the configuration variable `ACCOUNTS_USERINFO_HEADERS` is set to `True`. This makes the session and user id available to upstream servers like Nginx.

13.4.3 Minor changes in v3.1

Token expiration

The token expiration was changed from 5 days to 30 minutes for the password reset token and email confirmation token. Using the tokens will as a side-effect login in the user, which means that if the link is leaked (e.g. forwarded by the users themselves), then another person can use the link to access the account. Flask-Security v3.1.0 addresses this issue, but has not yet been released.

Globus.org OAuth Login

Invenio v3.1 now comes with support for login with your Globus.org account. The feature was contributed by University of Chicago.

See Invenio-OAuthClient for details.

Health-check view

A `/ping` view that can be enabled via the `APP_HEALTH_BLUEPRINT_ENABLED` configuration variable has been added to support load balancers like HAProxy to check if the application server is responsive.

13.4.4 Backwards incompatible changes

- **Pytest-Invenio:** The `celery_config` fixture has been renamed to `celery_config_ext` due to naming conflict with fixture provided by Celery.

13.4.5 Deprecations in v3.1

The following list of features have been deprecated and will be removed in either Invenio v3.2 or Invenio v3.3:

Elasticsearch v2 support

Elasticsearch v2 support will be removed in Invenio v3.2. Elasticsearch v2 has reached end of life and no longer receives any bug or security fixes.

Both the support in Invenio-Search for creating indexes for v2 as well as any v2 Elasticsearch mappings in other Invenio modules will be removed.

AMD/RequireJS

Invenio's assets build system based on AMD/RequireJS will be removed in Invenio v3.3.

This involves e.g. the two CLI commands:

```
$ invenio npm
$ invenio assets build
```

Several Python modules in Invenio-Assets will be removed, including (but not limited to):

- `invenio_assets.npm`
- `invenio_assets.filters`
- `invenio_assets.glob`
- `invenio_assets.proxies`

Also, bundle definitions in other Invenio modules will be removed. These are usually located in `bundles.py` files, e.g.:

- `invenio_theme.bundles`

Also, some static files will be removed from bundles, e.g.:

- `invenio_theme/static/js/*`
- `invenio_theme/static/scss/*`

DynamicPermission class

The `invenio_access.DynamicPermission` class will be removed in Invenio v3.2. It has been superseded by the `invenio_access.Permission` class. The `Permission` class by default deny an action in case no user/role is assigned. The `DynamicPermission` instead allowed an action if no user/role was assigned.

Records CLI

The following CLI commands will be removed in Invenio v3.2:

```
$ invenio records create
$ invenio records delete
$ invenio records patch
```

Please use the REST API instead to create, patch and delete records.

AngularJS (reminder from v3.0)

In Invenio v3.0 we deprecated the AngularJS 1.4 application Invenio-Search-JS as AngularJS by that time was already outdated. We have selected React and SemanticUI as the replacement framework for AngularJS.

The new Webpack build system released in Invenio v3.1 is part of the strategy to move from AngularJS to React (note however that you can use Webpack with your favorite framework, including AngularJS).

We have started the rewrite of Invenio-Search-JS and have already released the first version of React-SearchKit which eventually will replace Invenio-Search-JS.

13.4.6 Features removed in v3.1

The following already deprecated features have been removed in Invenio v3.1.

- `invenio_records.tasks` was removed from the Invenio-Records module.

13.4.7 Maintenance policy

Invenio v3.1 will be supported with bug and security fixes until the release of Invenio v3.3 and minimum until 2020-03-31.

See our *Maintenance Policy*.

13.4.8 What's next?

We originally planned to release the Files and Statistics bundle in Invenio v3.1. We however decided it was more urgent to release the new Webpack build system in order to avoid too much code being written against the old build system.

In Invenio v3.2 we are planning to release the **Files** bundle including:

- **invenio-files-rest**
  - Object storage REST API for Invenio with many supported backend storage protocols and file integrity checking.
- **invenio-iiif**

  - International Image Interoperability Framework (IIIF) server for making thumbnails and zooming images.
- invenio-previewer
  - Previewer for Markdown, JSON/XML, CSV, PDF, JPEG, PNG, TIFF, GIF and ZIP files.
- invenio-records-files
  - Integration layer between object storage and records.

13.5 Version 3.0.2

Released 2019-07-15

Invenio v3.0.2 fixes two security issues.

13.5.1 Security fixes

- **Invenio-App:** Fixed a Host header injection vulnerability.
- **Invenio-Records:** Fixed a Cross-Site Scripting vulnerability in the administration interface.

13.5.2 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.

13.6 Version 3.0.1

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

13.6.1 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-Continiuum.
- **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.

13.6.2 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.

13.6.3 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.

13.7 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.

13.7.1 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.

13.7.2 Getting started

In order to get started developing with Invenio v3 follow our *Quickstart*.

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

13.7.3 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+.

13.7.4 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.

13.7.5 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.

13.8 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

13.9 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

13.10 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

CHAPTER 14

Community

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

14.1 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 exsiting 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.

14.2 Communication channels

14.2.1 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.

14.2.2 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.

14.2.3 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.

14.2.4 Project website

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

14.2.5 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.

14.2.6 Twitter

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

- https://twitter.com/inveniosoftware

14.2.7 Mailing lists

The mailing lists are currently not very active and are primiarly 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.

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

14.3.1 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 *Link*.

- **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 *Quickstart*.
- **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

14.3.2 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.

14.3.3 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.

14.4 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.

14.4.1 Python

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

14.4.2 Indention style and whitespace

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

14.5 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.

14.5.1 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.

14.5.2 Working with Git and GitHub

There are a couple of utilities that allow you to work more efficiently with Git and GitHub.

Hub

Hub is a command-line wrapper for git that makes it easier to work with GitHub. See the installation instructions for how install Hub.

Here is a short overview of possibilities:

```
`console # Clone one of your personal repositories from GitHub $ git clone
invenio-app # Fetch the upstream inveniosoftware/invenio-app $ git fetch
inveniosoftware `
```

14.6 Translation guide

Invenio has been translated to more than 25 languages.

14.6.1 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.

14.6.2 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.

14.6.3 Developers

Please see dedicated invenio-i18n documentation.

14.7 Maintainer's guide

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

14.7.1 Overview

The goal of the maintainer system is to share the load of maintaining Invenio, spread Invenio experise 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 verfies 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 thoughout 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 alrady 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.

14.7.2 Setting up a repository

First, reach out to the Invenio architects and agree on scope and name of the new repository. The architects are their 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* exists 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 verifing 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 bootstraped using the cookiecutter-invenio-module.

JavaScript

JavaScript-based repositories must be boostraped using the generator-invenio-js-module.

14.7.3 Copyright and license policy

License

All released source code should by default be licensed under MIT License. Exceptions may be made to this policy on a case-by-case basis. Reasons for exceptions could be (but not limited to):

- Source code already released under another permissive license (e.g. Apache v2, BSD) or LGPL.

In all cases, source code should not be released under a copyleft license like GPL or AGPL.

Tracking copyright

Copyright statements

Copyright statements must be included in:

- `LICENSE` file in the root of the repository.
- Headers of all source code files.

Each copyright holder must have their own copyright statement:

```
Copyright (C) 2015-2018 CERN
Copyright (C) 2017 TIND
```

Copyright holders

Each copyright holder **must** be a legal entity.

Copyright is tracked for non-trivial contributions (i.e. creative work). By default we consider anything above 15 lines for a non-trivial contribution. Examples for which we do not track copyright is e.g. fixing a typo or tiny bug fixes.

Maintainer responsibility

Maintainers are responsible for asking each contributor who is the copyright holder of a given contribution (often, it's the employer who holds the copyright).

Attribution

Attribution is not tracked via copyright, but via the `AUTHORS` file.

Note: Legal entity: A legal entity can be human (physical persons) or non-human (juridical persons, e.g. corporations). A legal entity has privileges and obligations such as being able to enter into contracts, to sue or be sued. Thus, e.g. CERN is a legal entity but e.g. ATLAS, CMS and Invenio Collaboration are not considered legal entities by the law.

Explicit agreement

Contribtions only via GitHub

All contributions must be opened via pull requests on GitHub.

This way contributors have agreed to the GitHub Terms of Use, which states:

> "Whenever you make a contribution to a repository containing notice of a license, you license your contribution under the same terms, and you agree that you have the right to license your contribution under those terms."

This method avoid introducing either a Contributor License Agreement (CLAs) or a Developer Certificate of Origin.

14.8 Security policy

14.8.1 Reporting security issues

Normal bugs should be reported to the specific Invenio module's GitHub repository. However, due to sensitive nature of security issues, we ask that you do **not** report in a public fashion. This will allow us to distribute a security patch before potential attackers look at the issue.

If you believe you've found a security issue in Invenio, please send a description of the issue to info@inveniosoftware.org. Mails sent to this email address is logged in our request tracking system where Invenio architects have access to them.

You will first receive an automated notification from the request tracking system. Afterwards an Invenio architect will acknowledge the receipt (normally within 1-2 *working days*).

14.8.2 Supported versions

Please see our *Maintenance Policy*. Note that only supported versions are guaranteed to receive security fixes, and we only investigate if a given issue is affecting any of the currently supported versions of Invenio.

14.8.3 Disclosure of security issues

Advance notification

We will notify 2-5 days in advance about an upcoming security release and the severity level of the issue. The notification will not disclose any information about the issue except the severity level, and the sole purpose of the notification is to aid organisations to ensure they have staff available to handle the issue.

The notifications are sent to:

- Chatroom: https://gitter.im/inveniosoftware/invenio
- Mailing list: project-invenio-announce@cern.ch

Time-sensitive issues

In case the issue is particularly time-sensitive (e.g. known exploits in the wild) we may omit the advance notification.

Upstream libraries/frameworks

If an issue reported to us is affecting another library/framework we may report the issue privately to the maintainers of the affected library/framework.

Public announcement

On the day of the disclosure we take the following steps:

1. Apply patches to the Invenio source code
2. Issue new releases of Invenio and the affected modules to PyPI and/or NPM.
3. Notify the chatroom and mailing list (see above).
4. Post an entry to the Invenio blog.

14.8.4 Severity levels

We classify security issues according to the following severity levels:

- **Critical**
- **High**
- **Moderate**
- **Low**

The severity level is based on our self-calculated CVSS score for each specific vulnerability. CVSS is an industry standard vulnerability metric. You can learn more about CVSS at NIST NVD.

Credit

This security policy have drawn heavy inspiration from Django's security policy.

14.9 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 currrent 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.

14.9.1 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 progressive 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).

14.9.2 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.

14.9.3 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 limitied 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*.

14.9.4 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.

14.10 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.

14.11 License

CHAPTER 15

Build a module

Invenio modules are independent, interchangeable components that add functionalities. Each module exposes its own APIs and uses APIs of other modules. A full invenio application consists of a set of modules, and can be easily customized by adding or removing specific 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`.

The user interface and the REST API interface of a module may be split into separate modules, for example `invenio-records-ui` and `invenio-records-rest`, to clarify dependencies and offer an easier customization.

All modules have the same structure, which is defined in the cookiecutter-invenio-module template.

15.1 First steps

To create a new module, make sure you have cookiecutter installed and run the following command:

```
$ cookiecutter gh:inveniosoftware/cookiecutter-invenio-module
project_name [Invenio-FunGenerator]: Invenio-Foo
project_shortname [invenio-foo]:
package_name [invenio_foo]:
github_repo [inveniosoftware/invenio-foo]:
description [Invenio module that adds more fun to the platform.]:
author_name [CERN]:
author_email [info@inveniosoftware.org]:
year [2018]:
copyright_holder [CERN]:
copyright_by_intergovernmental [True]:
superproject [Invenio]:
```

(continues on next page)

(continued from previous page)

```
transifex_project [invenio-foo]:
extension_class [InvenioFoo]:
config_prefix [FOO]:
```

The newly scaffolded module will 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.

15.1.1 *.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.

15.1.2 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 several 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 can find the description of your module with the values entered in cookiecutter. At the end, you can find the `entrypoints` section.

15.1.3 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 any remaining checks.

15.1.4 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.

15.1.5 examples folder

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

15.1.6 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.

15.1.7 invenio_foo folder

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

The files that already exist are standard, and are covered 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).

MANIFEST.in

This file lists all the files included in the sub-folders. It should be updated before the first commit.

config.py

All configuration variables should be declared in this file.

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`.

version.py

File containing the version of your module.

views.py

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

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.

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.

15.2 Install a module

First of all, create a virtualenv for the module:

```
$ mkvirtualenv my_venv
```

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

```
(my_venv)$ cd invenio-foo/
(my_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.

If you need multiple options, you can chain them: `[tests,docs]`.

15.3 Run the tests

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

```
(my_venv)$ git init
(my_venv)$ git add --all
(my_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_venv)$ ./run-tests.sh
```

15.4 Build the documentation

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

```
(my_venv)$ python setup.py build_sphinx
```

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

15.5 Run the example application

The example application is a minimal application that presents the features of your module. The example application is useful during development for testing. By default, it simply prints a welcome page. To try it, go into the `examples` folder and run:

```
(my_venv)$ ./app-setup.sh
(my_venv)$ ./app-fixtures.sh
(my_venv)$ export FLASK_APP=app.py FLASK_DEBUG=1
(my_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 stopping the server.

15.6 Publishing on GitHub

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

First, create an empty repository in your GitHub account. Be sure not to 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:

```
$ git remote add origin URL-OF-YOUR-GITHUB-REPO
```

We can commit and push the generated files:

```
$ git commit -am "Initial module structure"
$ git push --set-upstream origin master
```

Finally, we create a new branch to develop on it.

```
$ git checkout -b dev
```

15.7 Use the module in your application

Integrating a new module to a full Invenio application comes down to adding it as a dependency in the central `Pipfile`. In order to do that, you should have published your module on GitHub and run the following command from the root folder of your Invenio application:

```
$ pipenv install URL-OF-YOUR-GITHUB-REPO
```

`pipenv` will update the `Pipfile` and install your module in the virtual enviroment of your application.

If your module has been released on PyPI, you can install it in your application by running the following command:

```
$ pipenv install invenio-foo
```

15.8 Next steps

To learn more about the development process in Invenio, follow the next guide *Developing with Invenio*.

CHAPTER 16

Developing with Invenio

As described in the previous section *Build a module*, Invenio has a modular design. Therefore in order to extend the application, we need to develop new modules. After covering how to create modules with cookiecutter, in this section we will go through the details of the development process.

16.1 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.

16.1.1 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.

16.1.2 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_foo/forms.py`

```python
"""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(
        'Description', [validators.DataRequired()]
    )
```

16.1.3 2. Create the views

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

- `create`: Form template
- `success`: Success template

and register all the views to our application.

```python
"""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_babelex 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_foo',
    __name__,
    template_folder='templates',
    static_folder='static',
)


@blueprint.route("/")
def index():
    """Basic view."""
    return render_template(
        "invenio_foo/index.html",
        module_name=_('invenio-foo'))


@blueprint.route('/create', methods=['GET', 'POST'])
```

(continues on next page)

(continued from previous page)

```python
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_foo.success'))

    records = _get_all()
    return render_template('invenio_foo/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_foo/success.html')
```

16.1.4 3. Create the templates

And now, let's create the templates.

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

```
{% extends config.FOO_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">
```

(continues on next page)

(continued from previous page)

```
          <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_foo.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 }}
              {{ 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_foo/templates/invenio_foo/* which will be rendered after a record is created.

```
{% extends config.FOO_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_foo.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>
```

(continues on next page)

(continued from previous page)

```
        </center>
      </div>
    </div>
  </div>
{% endblock page_body %}
```

16.1.5 4. Write 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_foo/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)
```

16.1.6 5. Create the custom-record JSON Schema

Our records can use a custom schema. To define and use this schema, we create the `custom-record-v1.0.0.json` file inside the `records` folder of your data model project (`my-datamodel` from the Quickstart tutorial *Understanding data models*).

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

16.2 Demo time

Let's now see our Invenio module in action after it has been integrated in our Invenio instance.

First, install the new invenio-foo module in the virtual enviroment of the Invenio instance:

```
$ pipenv shell  # activate the app virtual env
(my-site) $ cd ../invenio-foo
(my-site) $ pip install --editable .
```

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

```
(my-site) $ cd ../my-site
(my-site) $ ./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 in the `Title` field and click on submit: you will see a validation error on the form. Fill in the `Description` field and click on submit: the form is now valid and it navigates you to the `/success` page.