invenio-records Documentation

Release 1.0.1

CERN

Dec 17, 2018

Contents

Invenio-Records is a metadata storage module. A *record* is a JSON document with revision history identified by a unique UUID .

Features:

- Generic JSON document storage with revision history.

- JSONSchema validation of documents.

- Records creation, update and deletion.

- CLI and administration interface for CRUD operations on records.

Further documentation available Documentation: https://invenio-records.readthedocs.io/

User's Guide

This part of the documentation will show you how to get started in using Invenio-Records.

1.1 Installation

Invenio-Records can be installed from PyPI. Several installation options are possible, for example to use SQLite database backend:

```
pip install invenio-records[sqlite]
```

The other installation [options] include:

access for access control capabilities;

docs for documentation building dependencies;

mysql to use MySQL database backend;

postgresql to use PostgreSQL database backend;

sqlite to use SQLite database backend;

admin for Flask administration interfaces;

tests for test dependencies.

1.2 Usage

Invenio-Records is a metadata storage module.

In a few words, a record is basically a structured collection of fields and values (metadata) which provides information about other data.

A record (and each revision) is identified by a unique UUID, as most of the others entities in Invenio.

Invenio-Records is a core component of Invenio and it provides a way to create, update and delete records. Records are versioned, to keep track of modifications and to be able to revert back to a specific revision.

When creating or updating a record, if the record contains a schema definition, the record data will be validated against its schema. Moreover, data format can for each field be also validated.

When deleting a record, two options are available:

- **soft deletion**: record will be deletes but keeping its identifier and history, to ensure that the same record's identifier cannot be reused, and that older revisions can be retrieved.

- **hard deletion**: record will be completely deleted with its history.

Records creation and update can be validated if the schema is provided.

Records CRUD operations are available using the administration interface and the CLI, which also allows batch operations.

If InvenioPIDStore is installed, it also enables to mint PIDs in a record using the CLI.

Further documentation available Documentation: https://invenio-records.readthedocs.io/

1.2.1 Initialization

Create a Flask application:

```
>>> import os
>>> db_url = os.environ.get('SQLALCHEMY_DATABASE_URI', 'sqlite://')
>>> from flask import Flask
>>> app = Flask('myapp')
>>> app.config.update({
...     'SQLALCHEMY_DATABASE_URI': db_url,
...     'SQLALCHEMY_TRACK_MODIFICATIONS': False,
... })
```

Initialize Invenio-Records dependencies and Invenio-Records itself:

```
>>> from invenio_db import InvenioDB
>>> ext_db = InvenioDB(app)
>>> from invenio_records import InvenioRecords
>>> ext_records = InvenioRecords(app)
```

The following examples needs to run in a Flask application context, so let's push one:

```
>>> app.app_context().push()
```

Also, for the examples to work we need to create the database and tables (note, in this example we use an in-memory SQLite database by default):

```
>>> from invenio_db import db
>>> db.create_all()
```

1.2.2 CRUD operations

Creation

Let's **create** a very simple record:

```
>>> from invenio_records import Record
>>> record = Record.create({"title": "The title of the record"})
>>> db.session.commit()
>>> assert record.revision_id == 0
```

A new row has been added to the database, in the table `records_metadata`: this corresponds to the record metadata, first version (version 1).

Update

Let's try to **update** the previously created record with new data. This will create a new version of the previous with the same `uuid` but incremented version/revision id. Update the record and **commit** the changes to apply them to the record:

```
>>> record['title'] = 'The title of the 2nd version of the record'
>>> record = record.commit()  # validate new data and store changes
>>> db.session.commit()
>>> assert record.revision_id == 1
```

A second row has been added, version 2. You can access to the different versions by doing:

```
>>> rec_v1 = record.revisions[0]
>>> rec_v2 = record.revisions[1]
```

Reverting

To **restore** the first version of the record, just:

```
>>> record = record.revert(0)
>>> db.session.commit()
>>> assert record.revision_id == 2
```

Patch

It is also possible to **patch** a record to perform multiple operations in one shot:

```
>>> record = Record.create({"title": "First title"})
>>> db.session.commit()
>>> assert len(record.revisions) == 1
```

```
>>> ops = [
...     {"op": "replace", "path": "/title", "value": "Title first record"},
...     {"op": "add", "path": "/description", "value": "Record description"}
... ]
```

```
>>> record = record.patch(ops)
>>> record = record.commit()
>>> db.session.commit()
>>> assert len(record.revisions) == 2
```

See JSON Patch documentation to have nice examples.

Deletion

Let's create another record and then **soft delete** it:

```
>>> record = Record.create({"title": "Record to be deleted"})
>>> db.session.commit()
>>> record['title'] = 'Record to be deleted version 2'
>>> record = record.commit()
>>> db.session.commit()
```

```
>>> deleted = record.delete()
```

There is only one row left in the database corresponding to this record. Notice that the `json` column is empty, but the `uuid` is still there. This ensures uniqueness. The record can be retrieved by doing:

```
>>> deleted = Record.get_record(record.id, with_deleted=True)
>>> assert deleted.id == record.id
```

Let's **hard delete** it, completely:

```
>>> deleted = record.delete(force=True)
```

Now, try to retrieve it, it will throw an exception.

```
>>> Record.get_record(record.id,
...                    with_deleted=True)
Traceback (most recent call last):
  ...
NoResultFound: No row was found for one()
```

1.2.3 Record validation

When creating or updating a record, the input data can be validated to ensure that it is conform to a specified schema and values formats are respected. The validation is provided by the jsonschema library.

How `jsonschema` works

- **Format checker:** create a custom format checker (or use one of the available), for example to validate if the first letter of a string is uppercase:

  ```
  >>> from jsonschema import FormatChecker
  >>> from jsonschema.validators import Draft4Validator
  >>> checker = FormatChecker()
  >>> f = checker.checks("uppercaseFirstLetter")(lambda value: value[0]
  ...                                            .isupper())
  >>> validator = Draft4Validator({"format": "uppercaseFirstLetter"},
  ...                             format_checker=checker)
  ```

  Now, let's try it out:

  ```
  >>> validator.validate("Title of the record")
  ```

  Does not throw any exception, because the data is valid, the first letter is uppercase.

```
>>> validator.validate(
...                "title of the record")
Traceback (most recent call last):
  ...
ValidationError: 'title of the record' is not a 'uppercaseFirstLetter'
  ...
```

This raises a ValidationError error exception, because the first letter is lowercase.

- **Schema validator:** create a validator to ensure that the input data structure, fields and types conform to a specific schema.

```
>>> schema = {
...     'type': 'object',
...     'properties': {
...         'title': { 'type': 'string' },
...         'description': { 'type': 'string' }
...     },
...     'required': ['title']
... }
```

Try to validate a record without the field *title*, which is required.

```
>>> from jsonschema.validators import validate
>>> record = {"description": "Description but no title"}
>>> validate(record, schema)
Traceback (most recent call last):
  ...
ValidationError: 'title' is a required property
  ...
```

If the JSON schema is not defined inside the JSON itself, like in the example, but it is defined somewhere else (e.g. any schema provider service), the record should contain the $ref field with the URI link to the schema definition. Record provides a method *api.RecordBase.replace_refs()* that will resolve the URI in the $ref field and return a new Record with the schema definition injected.

Invenio-Records validation

Let's put everything together and create a record with validation and format checking: define a schema with a mandatory title field and a validation format for the title field.

```
>>> from jsonschema import FormatChecker
>>> checker = FormatChecker()
>>> f = checker.checks("uppercaseFirstLetter")(lambda value: value[0]
...                                             .isupper())
>>> schema = {
...     'type':'object',
...     'properties': {
...         'title': {
...             'type':'string',
...             'format': 'uppercaseFirstLetter'
...         },
...         'description': {
...             'type':'string'
...         }
...     },
```

```
...       'required': ['title']
... }
```

Create a new record with an invalid value format for the `title` field. Notice that the `schema` must be defined in the record with the field `$schema` and the format checker must be passed as `kwarg` argument with the key `format_checker`, to be taken into account by the *jsonschema* library.

```
>>> record = {
...       "$schema": schema,
...       "title": "title of this record",  # first letter is lowercase
...       "description": "Description of this record"
... }
>>> rec = Record.create(record,
...                 format_checker=checker)
Traceback (most recent call last):
  ...
ValidationError: 'title of this record' is not a 'uppercaseFirstLetter'
  ...
```

Create a new record without the `title` field:

```
>>> record = {
...       "$schema": schema,
...       "description": "Description of this record without a title"
... }
>>> rec = Record.create(record,
...                 format_checker=checker)
Traceback (most recent call last):
  ...
ValidationError: 'title' is a required property
  ...
```

1.2.4 CLI

The CLI provides a way of creating, patching or deleting records. Batch operations should be performed using the CLI.

Create a new record:

```
$ echo '{"title": "New record"}' | flask records create
```

Create multiple records:

```
$ echo '[{"title": "1st"},{"title":"2nd"}]' | flask records create
```

A file with a list of records can be specified as parameter to create multiple records in one shot. It is also possible to specify a list of `ids`, where each `id` corresponds to an input record, respecting the ordering.

In case of already existing `id`, the `force` parameter will create a new revision of the record:

```
$ echo '{"title": "New record"}' | flask records create \
  -i deadbeef-9fe4-43d3-a08f-38c2b309afba
$ echo '{"title": "Same new record"}' | flask records create --force \
  -i deadbeef-9fe4-43d3-a08f-38c2b309afba
```

Patch an existing record:

```
$ echo '{"title": "New record"}' | flask records create \
  -i deadbeef-9fe4-43d3-a08f-38c2b309afbe
$ echo '[{"op": "replace", "path": "/title", "value": "Patched"}]' | \
  flask records patch -i deadbeef-9fe4-43d3-a08f-38c2b309afbe
```

Soft and hard delete a record:

```
$ echo '{"title": "New record"}' | flask records create \
  -i 28c18220-f22e-480c-88ea-cd414aef035b
$ flask records delete -i 28c18220-f22e-480c-88ea-cd414aef035b
$ flask records delete --force -i 28c18220-f22e-480c-88ea-cd414aef035b
```

1.2.5 Minting PIDs

If the module InvenioPIDStore is installed and loaded, the CLI option `--pid-minter` allows minting PIDs in records.

To use `InvenioPIDStore`, initialize your app with:

```
>>> from invenio_pidstore import InvenioPIDStore
>>> ext_pid = InvenioPIDStore(app)
```

Then, when creating a record/records using the CLI, the name of an existing PID minter can be specified as parameter:

```
$ echo '{"title": "New record with PID"}' | flask records create \
  -i deadbeef-9fe4-43d3-a08f-38c2b309afbc --pid-minter recid
$ flask run
$ curl http://127.0.0.1:5000/deadbeef-9fe4-43d3-a08f-38c2b309afbc


    {
      "control_number": "1",
      "title": "New record with PID"
    }
```

See InvenioPIDStore documentation for more information.

1.2.6 Signals

Invenio-Records provides several types of signals and they can be used to react to events to read or modify data before or after an operation.

Events are sent in case of:

- record creation, before and after

- record update, before and after

- record deletion, before and after

- record revert, before and after

Let's modify the record before creation and verify, after creation, that the record has been correctly modified:

```
>>> from invenio_records.signals import (before_record_insert, \
...                                       after_record_insert)
>>> def before_record_creation_add_flag(sender, *args, **kwargs):
```

(continues on next page)

```
...        record = kwargs['record']
...        record['created_with'] = 'Invenio'
...
>>> listener = before_record_insert.connect(before_record_creation_add_flag)
>>> def after_record_creation(sender, *args, **kwargs):
...        record = kwargs['record']
...        assert 'created_with' in record
...
>>> listener = after_record_insert.connect(after_record_creation)
>>> rec_events = Record.create({"title": "My new record"})
>>> db.session.commit()
```

See *API Docs* for extensive API documentation.

1.3 Example application

Prepare the example app:

```
$ pip install -e .[all,sqlite]
$ cd examples
$ ./app-setup.sh
$ export FLASK_APP=app.py FLASK_DEBUG=1
$ flask run
```

Now, you can use *invenio-records*. Create a test record via CLI:

```
$ echo '{"title": "Test title"}' | flask records create \
  -i deadbeef-9fe4-43d3-a08f-38c2b309afba
```

Run the development server:

```
$ flask run
```

Retrieve a record via web:

```
$ curl http://127.0.0.1:5000/deadbeef-9fe4-43d3-a08f-38c2b309afba
```

To reset the example application run:

```
$ ./app-teardown.sh
```

See *Usage* for the extensive list of commands.

API Reference

If you are looking for information on a specific function, class or method, this part of the documentation is for you.

2.1 API Docs

Invenio module for metadata storage.

class invenio_records.ext.**InvenioRecords**(*app=None*, ***kwargs*)

Invenio-Records extension.

Extension initialization.

init_app(*app*, *entry_point_group='invenio_records.jsonresolver'*, ***kwargs*)

Flask application initialization.

> **Parameters**
>
> - **app** – The Flask application.
>
> - **entry_point_group** – The entrypoint for jsonresolver extensions. (Default: 'invenio_records.jsonresolver')

init_config(*app*)

Initialize configuration.

> **Parameters** **app** – The Flask application.

2.1.1 Record API

Record API.

class invenio_records.api.**Record**(*data*, *model=None*)

Define API for metadata creation and manipulation.

Initialize instance with dictionary data and SQLAlchemy model.

> **Parameters**
>
> - **data** – Dict with record metadata.
>
> - **model** – *RecordMetadata* instance.

commit(***kwargs*)

> Store changes of the current record instance in the database.
>
> 1. Send a signal *invenio_records.signals.before_record_update* with the current record to be committed as parameter.
>
> 2. Validate the current record data.
>
> 3. Commit the current record in the database.
>
> 4. **Send a signal *invenio_records.signals.after_record_update* with the committed record as parameter.**
>
> **Keyword Arguments**
>
> - **format_checker** – An instance of the class *jsonschema.FormatChecker*, which contains validation rules for formats. See *validate()* for more details.
>
> - **validator** – A *jsonschema.IValidator* class that will be used to validate the record. See *validate()* for more details.
>
> **Returns** The *Record* instance.

classmethod create(*data*, *id_=None*, ***kwargs*)

> Create a new record instance and store it in the database.
>
> 1. Send a signal *invenio_records.signals.before_record_insert* with the new record as parameter.
>
> 2. Validate the new record data.
>
> 3. Add the new record in the database.
>
> 4. Send a signal *invenio_records.signals.after_record_insert* with the new created record as parameter.
>
> **Keyword Arguments**
>
> - **format_checker** – An instance of the class *jsonschema.FormatChecker*, which contains validation rules for formats. See *validate()* for more details.
>
> - **validator** – A *jsonschema.IValidator* class that will be used to validate the record. See *validate()* for more details.
>
> **Parameters**
>
> - **data** – Dict with the record metadata.
>
> - **id** – Specify a UUID to use for the new record, instead of automatically generated.
>
> **Returns** A new *Record* instance.

delete(*force=False*)

> Delete a record.
>
> If *force* is `False`, the record is soft-deleted: record data will be deleted but the record identifier and the history of the record will be kept. This ensures that the same record identifier cannot be used twice, and

that you can still retrieve its history. If *force* is `True`, then the record is completely deleted from the database.

1. Send a signal *`invenio_records.signals.before_record_delete`* with the current record as parameter.

2. Delete or soft-delete the current record.

3. Send a signal *`invenio_records.signals.after_record_delete`* with the current deleted record as parameter.

> **Parameters** **`force`** – if `True`, completely deletes the current record from the database, otherwise soft-deletes it.
>
> **Returns** The deleted [`Record`](#) instance.

classmethod `get_record`(*id_*, *with_deleted=False*)
Retrieve the record by id.

Raise a database exception if the record does not exist.

> **Parameters**
>
> - **`id`** – record ID.
>
> - **`with_deleted`** – If *True* then it includes deleted records.
>
> **Returns** The [`Record`](#) instance.

classmethod `get_records`(*ids*, *with_deleted=False*)
Retrieve multiple records by id.

> **Parameters**
>
> - **`ids`** – List of record IDs.
>
> - **`with_deleted`** – If *True* then it includes deleted records.
>
> **Returns** A list of [`Record`](#) instances.

`patch`(*patch*)
Patch record metadata.

> **Params patch** Dictionary of record metadata.
>
> **Returns** A new [`Record`](#) instance.

`revert`(*revision_id*)
Revert the record to a specific revision.

1. Send a signal *`invenio_records.signals.before_record_revert`* with the current record as parameter.

2. Revert the record to the revision id passed as parameter.

3. Send a signal *`invenio_records.signals.after_record_revert`* with the reverted record as parameter.

> **Parameters** **`revision_id`** – Specify the record revision id
>
> **Returns** The [`Record`](#) instance corresponding to the revision id

`revisions`
Get revisions iterator.

---

class invenio_records.api.**RecordBase**(*data*, *model=None*)

> Base class for Record and RecordBase.

> Initialize instance with dictionary data and SQLAlchemy model.

> > **Parameters**
> >
> > > • **data** – Dict with record metadata.
> > >
> > > • **model** – *RecordMetadata* instance.

> **created**
>
> > Get creation timestamp.

> **dumps**(***kwargs*)
>
> > Return pure Python dictionary with record metadata.

> **id**
>
> > Get model identifier.

> **replace_refs**()
>
> > Replace the $ref keys within the JSON.

> **revision_id**
>
> > Get revision identifier.

> **updated**
>
> > Get last updated timestamp.

> **validate**(***kwargs*)
>
> > Validate record according to schema defined in $schema key.
> >
> > > **Keyword Arguments**
> > >
> > > > • **format_checker** – A format_checker is an instance of class jsonschema.
> > > >   FormatChecker containing business logic to validate arbitrary formats. For example:
> > > >
> > > >   ```
> > > >   >>> from jsonschema import FormatChecker
> > > >   >>> from jsonschema.validators import validate
> > > >   >>> checker = FormatChecker()
> > > >   >>> checker.checks('foo')(lambda el: el.startswith('foo'))
> > > >   <function <lambda> at ...>
> > > >   >>> validate('foo', {'format': 'foo'}, format_checker=checker)
> > > >   ```
> > > >
> > > >   returns None, which means that the validation was successful, while
> > > >
> > > >   ```
> > > >   >>> validate('bar', {'format': 'foo'},
> > > >   ...     format_checker=checker)
> > > >   Traceback (most recent call last):
> > > >     ...
> > > >   ValidationError: 'bar' is not a 'foo'
> > > >     ...
> > > >   ```
> > > >
> > > >   raises a jsonschema.exceptions.ValidationError.
> > > >
> > > > • **validator** – A jsonschema.IValidator class used for record validation. It will be
> > > >   used as *cls* argument when calling jsonschema.validate(). For example
> > > >
> > > >   ```
> > > >   >>> from jsonschema.validators import extend, Draft4Validator
> > > >   >>> NoRequiredValidator = extend(
> > > >   ...     Draft4Validator,
> > > >   ...     validators={'required': lambda v, r, i, s: None}
> > > >   ```

(continues on next page)

```
...    )
>>> schema = {
...        'type': 'object',
...        'properties': {
...            'name': { 'type': 'string' },
...            'email': { 'type': 'string' },
...            'address': {'type': 'string' },
...            'telephone': { 'type': 'string' }
...        },
...        'required': ['name', 'email']
... }
>>> from jsonschema.validators import validate
>>> validate({}, schema, NoRequiredValidator)
```

returns `None`, which means that the validation was successful, while

```
>>> validate({}, schema)
Traceback (most recent call last):
...
ValidationError: 'name' is a required property
...
```

raises a `jsonschema.exceptions.ValidationError`.

class `invenio_records.api.`**`RecordRevision`**(*model*)
   API for record revisions.

   Initialize instance with the SQLAlchemy model.

class `invenio_records.api.`**`RevisionsIterator`**(*model*)
   Iterator for record revisions.

   Initialize instance with the SQLAlchemy model.

   `next`()
      Python 2.7 compatibility.

2.1.2 CLI

Click command-line interface for record management.

`invenio_records.cli.`**`option_pid_minter`**(*_*)
   Empty option.

`invenio_records.cli.`**`records = <click.core.Group object>`**
   Records management.

`invenio_records.cli.`**`create = <click.core.Command object>`**
   Create new bibliographic record(s).

`invenio_records.cli.`**`patch = <click.core.Command object>`**
   Patch existing bibliographic record.

`invenio_records.cli.`**`delete = <click.core.Command object>`**
   Delete bibliographic record(s).

2.1.3 Configuration

Default values for records configuration.

`invenio_records.config.`**`RECORDS_VALIDATION_TYPES = {}`**
    Pass additional types when validating a record against a schema. For more details, see: https://python-jsonschema.readthedocs.io/en/latest/validate/#validating-types.

2.1.4 Errors

Errors for Invenio-Records module.

exception `invenio_records.errors.`**`MissingModelError`**
    Error raised when a record has no model.

exception `invenio_records.errors.`**`RecordsError`**
    Base class for errors in Invenio-Records module.

2.1.5 Models

Record models.

class `invenio_records.models.`**`RecordMetadata`**(**kwargs*)
    Represent a record metadata.

    The RecordMetadata object contains a `created` and a `updated` properties that are automatically updated.

    A simple constructor that allows initialization from kwargs.

    Sets attributes on the constructed instance using the names and values in `kwargs`.

    Only keys that are present as attributes of the instance's class are allowed. These could be, for example, any mapped columns or relationships.

    id
        Record identifier.

    json
        Store metadata in JSON format.

        When you create a new `Record` the `json` field value should never be `NULL`. Default value is an empty dict. `NULL` value means that the record metadata has been deleted.

    version_id
        Used by SQLAlchemy for optimistic concurrency control.

2.1.6 Signals

Record module signals.

`invenio_records.signals.`**`after_record_delete = <blinker.base.NamedSignal object at 0x7f0a2e`**
    Signal sent after a record is deleted.

    When implementing the event listener, the record data can be retrieved from *kwarg['record']*.

Note: Do not perform any modification to the record here: they will be not persisted.

---

invenio_records.signals.**after_record_insert = <blinker.base.NamedSignal object at 0x7f0a2e**
Signal sent after a record is inserted.

When implementing the event listener, the record data can be retrieved from *kwarg['record']*.

Note: Do not perform any modification to the record here: they will be not persisted.

---

invenio_records.signals.**after_record_revert = <blinker.base.NamedSignal object at 0x7f0a2e**
Signal sent after a record is reverted.

When implementing the event listener, the record data can be retrieved from *kwarg['record']*.

Note: Do not perform any modification to the record here: they will be not persisted.

---

invenio_records.signals.**after_record_update = <blinker.base.NamedSignal object at 0x7f0a2e**
Signal sent after a record is updated.

When implementing the event listener, the record data can be retrieved from *kwarg['record']*.

Note: Do not perform any modification to the record here: they will be not persisted.

---

invenio_records.signals.**before_record_delete = <blinker.base.NamedSignal object at 0x7f0a2e**
Signal is sent before a record is deleted.

When implementing the event listener, the record data can be retrieved from *kwarg['record']*.

invenio_records.signals.**before_record_insert = <blinker.base.NamedSignal object at 0x7f0a2e**
Signal is sent before a record is inserted.

When implementing the event listener, the record data can be retrieved from *kwarg['record']*. Example event
listener (subscriber) implementation:

```python
def listener(sender, *args, **kwargs):
    record = kwargs['record']
    # do something with the record

from invenio_records.signals import before_record_insert
before_record_insert.connect(listener)
```

invenio_records.signals.**before_record_revert = <blinker.base.NamedSignal object at 0x7f0a2e**
Signal is sent before a record is reverted.

When implementing the event listener, the record data can be retrieved from *kwarg['record']*.

invenio_records.signals.**before_record_update = <blinker.base.NamedSignal object at 0x7f0a2e**
Signal is sent before a record is updated.

When implementing the event listener, the record data can be retrieved from *kwarg['record']*.

Additional Notes

Notes on how to contribute, legal information and changes are here for the interested.

3.1 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

3.1.1 Types of Contributions

Report Bugs

Report bugs at https://github.com/inveniosoftware/invenio-records/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it.

Write Documentation

Invenio-Records could always use more documentation, whether as part of the official Invenio-Records docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/inveniosoftware/invenio-records/issues.

If you are proposing a feature:

- Explain in detail how it would work.

- Keep the scope as narrow as possible, to make it easier to implement.

- Remember that this is a volunteer-driven project, and that contributions are welcome :)

3.1.2 Get Started!

Ready to contribute? Here's how to set up *invenio-records* for local development.

1. Fork the *inveniosoftware/invenio-records* repo on GitHub.

2. Clone your fork locally:

   ```
   $ git clone git@github.com:your_name_here/invenio-records.git
   ```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

   ```
   $ mkvirtualenv invenio-records
   $ cd invenio-records/
   $ pip install -e .[all]
   ```

4. Create a branch for local development:

   ```
   $ git checkout -b name-of-your-bugfix-or-feature
   ```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass tests:

   ```
   $ ./run-tests.sh
   ```

   The tests will provide you with test coverage and also check PEP8 (code style), PEP257 (documentation), flake8 as well as build the Sphinx documentation and run doctests.

6. Commit your changes and push your branch to GitHub:

   ```
   $ git add .
   $ git commit -s
       -m "component: title without verbs"
       -m "* NEW Adds your new feature."
       -m "* FIX Fixes an existing issue."
       -m "* BETTER Improves and existing feature."
       -m "* Changes something that should not be visible in release notes."
   $ git push origin name-of-your-bugfix-or-feature
   ```

7. Submit a pull request through the GitHub website.

3.1.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests and must not decrease test coverage.

2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring.

3. The pull request should work for Python 2.7, 3.3, 3.4 and 3.5. Check https://travis-ci.org/inveniosoftware/invenio-records/pull_requests and make sure that the tests pass for all supported Python versions.

3.2 Changes

Version 1.0.1 (released 2018-12-14)

- Fix CliRunner exceptions
- Fix json schema url
- MIT license and shield badge

Version 1.0.0 (released 2018-03-23)

- Initial public release.

3.3 License

MIT License

Copyright (C) 2015-2018 CERN.

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

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

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

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

3.4 Contributors

- Alizee Pace

- Diego Rodriguez Rodriguez
- Esteban J. G. Gabancho
- Harris Tzovanakis
- Jacopo Notarstefano
- Jan Aage Lavik
- Javier Delgado
- Javier Martin Montull
- Jiri Kuncar
- Jose Benito Gonzalez Lopez
- Krzysztof Nowak
- Lars Holm Nielsen
- Leonardo Rossi
- Nicola Tarocco
- Nicolas Harraudeau
- Orestis Melkonian
- Paulina Lach
- Rémi Ducceschi
- Sami Hiltunen
- Tibor Simko

Python Module Index

i

Index

A

after_record_delete (in module invenio_records.signals), [16](#)

after_record_insert (in module invenio_records.signals), [16](#)

after_record_revert (in module invenio_records.signals), [17](#)

after_record_update (in module invenio_records.signals), [17](#)

B

before_record_delete (in module invenio_records.signals), [17](#)

before_record_insert (in module invenio_records.signals), [17](#)

before_record_revert (in module invenio_records.signals), [17](#)

before_record_update (in module invenio_records.signals), [17](#)

C

commit() (invenio_records.api.Record method), [12](#)

create (in module invenio_records.cli), [15](#)

create() (invenio_records.api.Record class method), [12](#)

created (invenio_records.api.RecordBase attribute), [14](#)

D

delete (in module invenio_records.cli), [15](#)

delete() (invenio_records.api.Record method), [12](#)

dumps() (invenio_records.api.RecordBase method), [14](#)

G

get_record() (invenio_records.api.Record class method), [13](#)

get_records() (invenio_records.api.Record class method), [13](#)

I

id (invenio_records.api.RecordBase attribute), [14](#)

id (invenio_records.models.RecordMetadata attribute), [16](#)

init_app() (invenio_records.ext.InvenioRecords method), [11](#)

init_config() (invenio_records.ext.InvenioRecords method), [11](#)

invenio_records (module), [3](#)

invenio_records.api (module), [11](#)

invenio_records.cli (module), [15](#)

invenio_records.config (module), [16](#)

invenio_records.errors (module), [16](#)

invenio_records.ext (module), [11](#)

invenio_records.models (module), [16](#)

invenio_records.signals (module), [16](#)

InvenioRecords (class in invenio_records.ext), [11](#)

J

json (invenio_records.models.RecordMetadata attribute), [16](#)

M

MissingModelError, [16](#)

N

next() (invenio_records.api.RevisionsIterator method), [15](#)

O

option_pid_minter() (in module invenio_records.cli), [15](#)

P

patch (in module invenio_records.cli), [15](#)

patch() (invenio_records.api.Record method), [13](#)

R

Record (class in invenio_records.api), [11](#)

RecordBase (class in invenio_records.api), [13](#)

RecordMetadata (class in invenio_records.models), [16](#)

RecordRevision (class in invenio_records.api), [15](#)

records (in module invenio_records.cli), [15](#)