invenio-db Documentation

Release 1.0.5

CERN

May 11, 2020

Contents

Database management for Invenio. Further documentation available on https://invenio-db.readthedocs.io/

CHAPTER 1

User's Guide

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

1.1 Installation

The Invenio-DB package is on PyPI so all you need is:

```
$ pip install invenio-db
```

Invenio-DB depends on:

SQLAlchemy-Utils
Flask-SQLAlchemy
Flask-Alembic

1.2 Configuration

The default values of configuration options are guessed based on installed packages.

SQLALCHEMY_DATABASE_URI
The database URI that should be used for the connection. Defaults to `'sqlite:///<instance_path>/<app.name>.db'`.

SQLALCHEMY_ECHO
Enables debug output containing database queries. Defaults to `True` if application is in debug mode (`app.debug == True`).

DB_VERSIONING
Enables versioning support using SQLAlchemy-Continuum. Defaults to `True` if `sqlalchemy_continuum` package is installed.

DB_VERSIONING_USER_MODEL
User class used by versioning manager. Defaults to `'User'` if `invenio_accounts` package is installed.

ALEMBIC
Dictionary containing general configuration for Flask-Alembic. It contains defaults for following keys:

- `'script_location'` points to location of the migrations directory. It is **required** key and defaults to location of `invenio_db.alembic` package resource.
- `'version_locations'` lists location of all independent named branches specified by Invenio packages in `invenio_db.alembic` entry point group.

Please check following packages for further configuration options:

1. Flask-SQLAlchemy
2. Flask-Alembic
3. SQLAlchemy-Continuum

1.3 Usage

Database management for Invenio.

First make sure you have Flask application with Click support (meaning Flask 0.11+):

```
>>> from flask import Flask
>>> app = Flask('myapp')
```

Next, initialize your extension:

```
>>> from invenio_db import InvenioDB
>>> db = InvenioDB(app)
```

1.3.1 Command-line interface

Invenio-DB installs the `db` command group on your application with the following commands:

- `create` - Create database tables.
- `drop` - Drop database tables.
- `init` - Initialize database.
- `destroy` - Destroy database.

and `alembic` command group for managing upgrade recipes:

- `branches` - Show branch points.
- `current` - Show current revision.
- `downgrade` - Run downgrade migrations.
- `heads` - Show latest revisions.
- `log` - Show revision log.
- `merge` - Create merge revision.
- `mkdir` - Make migration directory.

- `revision` - Create new migration.
- `show` - Show the given revisions.
- `stamp` - Set current revision.
- `upgrade` - Run upgrade migrations.

For more information about how to setup alembic revisions in your module, please read the section about *alembic*.

1.3.2 Models

Database models are created by inheriting from the declarative base `db.Model`:

```
# models.py
from invenio_db import db

class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(80), unique=True)
    email = db.Column(db.String(120), unique=True)
```

1.3.3 Setuptools integration

In order for the CLI commands to be aware of your models and upgrade recipies, you must either import your models in the application factory, or better simply specify the entry point item in `invenio_db.models` group. Invenio-DB then takes care of loading the models automatically. Alembic configuration of version locations is assembled from `invenio_db.alembic` entry point group.

```
# setup.py
# ...
setup(
    entry_points={
        'invenio_db.alembic': [
            'branch_name = mymodule:alembic',
        ],
        'invenio_db.models': [
            'mymodule = mymodule.models',
        ],
    },
)
```

1.4 Alembic for Invenio

Alembic is a database migration library used with SQLAlchemy ORM. Invenio works with the Flask-Alembic library, its documentation can be found here: http://flask-alembic.readthedocs.io/en/latest/

Invenio-DB fully supports alembic and each Invenio module having a database model is also expected to provide the corresponding alembic revisions. Alembic migrations do not work with SQLite.

1.4.1 Adding alembic support to existing modules

The following procedures assume `invenio_foo` is the name of the module for which you are adding alembic support.

The first step would be to add the entrypoint `invenio_db.alembic` in your `invenio_foo` setup.py as follows:

```
setup(
    ...
    entry_points={
        ...
        'invenio_db.alembic': [
            'invenio_foo = invenio_foo:alembic',
        ]
})
```

This will register the `invenio_foo/alembic` directory in alembic's `version_locations`.

Each module should create a branch for its revisions. In order to create a new branch, and in consequence the first revision, one should run:

```
$ invenio alembic revision "Create foo branch." -b invenio_foo -p <parent-revision> -
↪d <dependencies> --empty
```

-b sets the branch label (conventionally the name of the module)

-p sets the parent revision, as default all branches should root from the revision `dbdbc1b19cf2` in invenio-db

-d sets the dependencies if they exist. For example when there is a foreign key pointing to the table of another invenio module, we need to make sure that table exists before applying this revision, so we add the necessary revision tag as a dependency.

The second revision typically has the message "Create foo tables." and will create the tables defined in the models. This can be created following the procedure below.

1.4.2 Creating a new revision

After making changes to the models of a module, we need to create a new alembic revision so we are able to apply these changes in the DB during a migration. Firstly, to make sure that the DB is up to date we apply any pending revisions with:

```
$ invenio alembic upgrade heads
```

and now we can create the new revision with:

```
$ invenio alembic revision "Revision message." --path ``invenio_foo/alembic``
```

A short message describing the changes is required and the path parameter should point to the alembic directory of the module. If the path is not given the new revision will be placed in the invenio_db/alembic directory and should be moved.

1.4.3 Show current state

To see the list of revisions in the order they will be applied, run:

```
$ invenio alembic log
```

The list of heads for all branches is given by:

```
$ invenio alembic heads
```

in this list, revisions will be labeled as `(head)` or `(effective head)`. The difference being that effective heads are not shown in the `alembic_version` table in your database. As they are dependencies of other branches, they will be overwritten. `alembic_version` is a table created by alembic to keep the current revision state.

The list of the revisions that have been applied to the current database can be seen with:

```
$ invenio alembic current
```

1.4.4 Enabling alembic migrations in existing invenio instances

In order to integrate alembic when there is already a DB in place, we have to create an `alembic_version` table stamped with the revisions matching the current state of the DB:

```
$ invenio alembic stamp
```

Assuming that there have been no changes in the DB, and the models match the alembic revisions, alembic upgrades and downgrades will be working now.

Note that if there are any unnamed constraints, they will get the default names from the DB which can be different from the ones in the alembic revisions.

1.4.5 Naming Constraints

In http://alembic.zzzcomputing.com/en/latest/naming.html, the need for naming constraints in the models is explained. In invenio-db the '35c1075e6360' revision applies the naming convention for invenio. If models contain constraints that are unnamed an `InvalidRequestError` will be raised.

The naming convention rules are:

index	'ix_%(column_0_label)s'
unique	'uq_%(table_name)s_%(column_0_name)s'
check	'ck_%(table_name)s_%(constraint_name)s'
foreign key	'fk_%(table_name)s_%(column_0_name)s_%(referred_table_name)s'
primary key	'pk_%(table_name)s'

The constraints that produce a name longer that 64 characters will have to be named explicitly to a truncated form.

1.4.6 Testing revisions

When initially creating alembic revisions one has to provide a test case for them.

The test for the created revisions starts from an empty DB, upgrades to the last branch revision and then downgrades to the base. We can check that there are no discrepancies in the state of the DB between the revisions and the models, by asserting that alembic.compare_metadata() returns an empty list. An example can be found here: test_app.py#L130

1.5 Database session management

Invenio uses SQLAlchemy Toolkit and Object Relational Mapper for all database related operations.

1.5.1 Transactions are tied to requests

In Invenio, a transaction is tied to an HTTP request. By default, the transaction is automatically rolled back unless you explicitly call `db.session.commit()`.

1.5.2 Exceptions cause rollback

If an exception occurs during request handling, then the entire transaction will be rolled back unless there has been an explicit call to `db.session.commit()` before the exception was raised. This is because the default behavior is to rollback.

1.5.3 Why are transactions tied to requests?

Transactions are tied to requests, because the outer view, in charge of handling the request, needs full control of when a transaction is committed. If the view was not in charge, you could end up with inconsistent data in the database - for instance persistent identifier may have been committed, but the associated record was not committed. That is why Invenio makes use of SQLAlchemy's version counter feature to provide optimistic concurrency control (OCC) on the records table when the database transaction isolation level is below repeatable read isolation level.

1.5.4 When are SQL statements sent to the database server?

SQLAlchemy only sends the SQL statements (INSERT, UPDATE, SELECT, . . .) to the database server when needed, or when explicitly requested via e.g. `db.session.flush()` or `db.session.commit()`.

This means that in many cases, SQL INSERT and UPDATE statements are not sent to the server until a commit is called.

1.5.5 What about nested transactions?

Nested transactions are using database save points, which allow you to do a partial rollback. Also, nested transactions cause a flush to the database, meaning that the SQL statements are sent to the server.

1.5.6 When are partial rollbacks useful?

Partial rollbacks can be useful for instance if you want to try to insert a user, but the user already exists in the table. Then you can rollback the insert and instead execute an update statement at the application level.

1.5.7 When is flushing useful?

Explicitly forcing the flush of SQL statements to the database can be useful if you need a value from the database (e.g. auto-incrementing primary keys), and the application needs the primary key to continue. Also, they can be useful to force integrity constraints to be checked by the database, which may be needed by the database.

1.5.8 What happens with exceptions in nested transactions?

If an exception occurs in a nested transaction, first the save point will be rolled back, and afterwards the entire transaction will be rolled back unless the exception is caught.

For instance in the following code example, the entire transaction will be rolled back:

```
@app.route('/')
def index():
    # db operations 1 ....
    with db.session.begin_nested():
        # db operations 2 ....
        raise Exception()
    db.session.commit()
```

On the other hand, in the following example, the propagation of the exception is stopped, and only the db operations 2 are rolled back, while db operations 1 are committed to the database.

```
@app.route('/')
def index():
    # db operations 1 ....
    try:
        with db.session.begin_nested():
            # db operations 2 ....
        raise Exception()
        db.session.commit()
    except Exception:
        db.session.rollback()
    db.session.commit()
```

CHAPTER 2

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

Database management for Invenio.

class invenio_db.ext.**InvenioDB**(*app=None*, ***kwargs*)
Invenio database extension.

Extension initialization.

init_app(*app*, ***kwargs*)
Initialize application object.

init_db(*app*, *entry_point_group='invenio_db.models'*, ***kwargs*)
Initialize Flask-SQLAlchemy extension.

init_versioning(*app*, *database*, *versioning_manager=None*)
Initialize the versioning support using SQLAlchemy-Continuum.

Shared database object for Invenio.

invenio_db.shared.**NAMING_CONVENTION = {'ck': 'ck_%(table_name)s_%(constraint_name)s', 'fk**
Configuration for constraint naming conventions.

class invenio_db.shared.**SQLAlchemy**(*app=None*, *use_native_unicode=True*, *session_options=None*, *metadata=None*, *query_class=<class 'flask_sqlalchemy.BaseQuery'>*, *model_class=<class 'flask_sqlalchemy.model.Model'>*, *engine_options=None*)
Implement or overide extension methods.

apply_driver_hacks(*app*, *info*, *options*)
Call before engine creation.

`invenio_db.shared.`**`db = <SQLAlchemy engine=None>`**

Shared database instance using Flask-SQLAlchemy extension.

This object is initialized during initialization of `InvenioDB` extenstion that takes care about loading all entrypoints from key `invenio_db.models`.

`invenio_db.shared.`**`do_sqlite_begin`**`(`*dbapi_connection*`)`

Ensure SQLite transaction are started properly.

For further details see "Foreign key support" sections on https://docs.sqlalchemy.org/en/rel_1_0/dialects/sqlite.html#pysqlite-serializable # noqa

`invenio_db.shared.`**`do_sqlite_connect`**`(`*dbapi_connection*, *connection_record*`)`

Ensure SQLite checks foreign key constraints.

For further details see "Foreign key support" sections on https://docs.sqlalchemy.org/en/latest/dialects/sqlite.html#foreign-key-support

`invenio_db.shared.`**`metadata = MetaData(bind=None)`**

Default database metadata object holding associated schema constructs.

Click command-line interface for database management.

`invenio_db.cli.`**`abort_if_false`**`(`*ctx*, *param*, *value*`)`

Abort command is value is False.

CHAPTER 3

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-db/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-DB could always use more documentation, whether as part of the official Invenio-DB 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-db/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-db* for local development.

1. Fork the *inveniosoftware/invenio-db* repo on GitHub.
2. Clone your fork locally:

   ```
   $ git clone git@github.com:your_name_here/invenio-db.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-db
   $ cd invenio-db/
   $ 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.com/inveniosoftware/invenio-db/pull_requests and make sure that the tests pass for all supported Python versions.

3.2 Changes

Version 1.0.5 (released 2020-05-11)

- Deprecated Python versions lower than 3.6.0. Now supporting 3.6.0 and 3.7.0
- Use centrally managed Flask version (through Invenio-Base)
- Bumped SQLAlchemy version to `>=1.1.0`
- SQLAlchemy-Utils set to `<0.36` due to breaking changes with MySQL (`VARCHAR` length)
- Enriched documentation on DB session management
- Stop using example app

Version 1.0.4 (released 2019-07-29)

- Unpin sqlalchemy-continuum
- Added tests for postgresql 10

Version 1.0.3 (released 2019-02-22)

- Added handling in case of missing Sqlite db file.

Version 1.0.2 (released 2018-06-22)

- Pin SQLAlchemy-Continuum.

Version 1.0.1 (released 2018-05-16)

- Minor fixes in documenation links and the license file.

Version 1.0.0 (released 2018-03-23)

- Initial public release.

3.3 License

3.4 Contributors

- Alexander Ioannidis
- Alizee Pace
- Dinos Kousidis
- Esteban J. G. Gabancho
- Jacopo Notarstefano
- Jiri Kuncar
- Jose Benito Gonzalez Lopez
- Lars Holm Nielsen
- Leonardo Rossi
- Nicolas Harraudeau
- Paulina Lach
- Sami Hiltunen
- Tibor Simko

Python Module Index

Index