Flask-Continuum Documentation

Release 0.1.7

Blake Printy

Apr 29, 2020

Contents

CHAPTER 1

Overview

Flask-Continuum is a lightweight Flask extension providing data provenance and versioning support to Flask applications using SQLAlchemy. It is built on top of the sqlalchemy-continuum package, and provides a more Flask-y development experience for app configuration. If you'd like to configure your application with `sqlalchemy-continuum` directly, consult the `sqlalchemy-continuum` documentation.

1.1 A Minimal Application

Setting up the flask application with extensions:

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_continuum import Continuum

app = Flask(__name__)
db = SQLAlchemy(app)
continuum = Continuum(app, db)
```

Or, using via the Flask app factory pattern:

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_continuum import Continuum

db = SQLAlchemy()
continuum = Continuum(db=db)
app = Flask(__name__)
db.init_app(app)
continuum.init_app(app)
```

The following is a minimal example highlighting how the extension is used. Much of the example was taken from the SQLAlchemy-Continuum documentation to show how this plugin extends that package for a Flask application:

```
from flask_continuum import VersioningMixin

# defining database schema
class Article(db.Model, VersioningMixin):
    __tablename__ = 'article'

    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    name = db.Column(db.Unicode(255))
    content = db.Column(db.UnicodeText)


# later in api or request handlers
article = Article(name='Some article', content='Some content')
session.add(article)
session.commit()

# article has now one version stored in database
article.versions[0].name
# 'Some article'

article.name = 'Updated name'
session.commit()

article.versions[1].name
# 'Updated name'


# lets revert back to first version
article.versions[0].revert()

article.name
# 'Some article'
```

For more in-depth discussion on design considerations and how to fully utilize the plugin, see the User Guide.

CHAPTER 2

User Guide

2.1 Overview

Flask-Continuum is a lightweight Flask extension providing data provenance and versioning support to Flask applications using SQLAlchemy. It is built on top of the sqlalchemy-continuum package, and provides a more Flask-y development experience for app configuration. If you'd like to configure your application with `sqlalchemy-continuum` directly, consult the `sqlalchemy-continuum` documentation.

2.1.1 A Minimal Application

Setting up the flask application with extensions:

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_continuum import Continuum

app = Flask(__name__)
db = SQLAlchemy(app)
continuum = Continuum(app, db)
```

Or, using via the Flask app factory pattern:

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_continuum import Continuum

db = SQLAlchemy()
continuum = Continuum(db=db)
app = Flask(__name__)
db.init_app(app)
continuum.init_app(app)
```

The following is a minimal example highlighting how the extension is used. Much of the example was taken from the SQLAlchemy-Continuum documentation to show how this plugin extends that package for a Flask application:

```
from flask_continuum import VersioningMixin

# defining database schema
class Article(db.Model, VersioningMixin):
    __tablename__ = 'article'

    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    name = db.Column(db.Unicode(255))
    content = db.Column(db.UnicodeText)


# later in api or request handlers
article = Article(name='Some article', content='Some content')
session.add(article)
session.commit()

# article has now one version stored in database
article.versions[0].name
# 'Some article'

article.name = 'Updated name'
session.commit()

article.versions[1].name
# 'Updated name'


# lets revert back to first version
article.versions[0].revert()

article.name
# 'Some article'
```

For more in-depth discussion on design considerations and how to fully utilize the plugin, see the User Guide.

2.2 Installation

To install the latest stable release via pip, run:

```
$ pip install Flask-Plugin
```

Alternatively with easy_install, run:

```
$ easy_install Flask-Plugin
```

To install the bleeding-edge version of the project:

```
$ git clone http://github.com/bprinty/Flask-Plugin.git
$ cd Flask-Plugin
$ python setup.py install
```

2.3 Usage

The sections below detail how to fully use this module, along with context for design decisions made during development of the plugin.

2.3.1 Setup

Obviously, this plugin requires the use of SQLAlchemy for model definitions. However, there are two common patterns for how SQLAlchemy models are configured for a Flask application:

1. Using the Flask-SQLAlchemy plugin for simplifying boilerplate associated with configuring a SQLAlchemy-backed Flask application (recommended).
2. Using SQLAlchemy directly with the declarative system for defining models in your application.

If you're using the `Flask-SQLAlchemy` plugin, you can configure this plugin by passing the `db` parameter into the extension:

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_continuum import Continuum

db = SQLAlchemy()
continuum = Continuum(db=db)
app = Flask(__name__)
db.init_app(app)
continuum.init_app(app)
```

If you're using SQLAlchemy directly, you need to pass the SQLAlchemy `engine` to the plugin. See the SQLAlchemy documentation for more context on setting up the `engine`:

```
from flask import Flask
from sqlalchemy import create_engine
from flask_continuum import Continuum

engine = create_engine('postgresql://admin:password@localhost:5432/my-database')
continuum = Continuum(engine=engine)
app = Flask(__name__)
continuum.init_app(app)
```

Aside from the plugin configuration detailed above, there is no additional steps required for configuring mappers or setting up `sqlalchemy-continuum`. SQLAlchemy mappers for versioning tables will be set up when the first connection to the application database is made. For more information on additional configuration options, see the *Other Customizations* section below.

2.3.2 Mixins

In order to add versioning support to models in your application, you can either:

1. Use the `VersioningMixin` from this package to add versioning support and additional helper methods (recommended).
2. Add a `__versioned__ = {}` property to model classes.

With the `VersioningMixin`, you can add versioning to a model via:

```
class Article(db.Model, VersioningMixin):
    __tablename__ = 'article'

    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    name = db.Column(db.Unicode(255))
    content = db.Column(db.UnicodeText)
    updated_at = db.Column(db.DateTime, default=datetime.now)
    created_at = db.Column(db.DateTime, onupdate=datetime.now)
```

Additionally, if you only want to track specific fields in the database (for more efficient changeset processing), you can use the following syntax:

```
class Article(db.Model, VersioningMixin):
    __versioned__ = {
        'include': ['name', 'content']
    }
    __tablename__ = 'article'

    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    name = db.Column(db.Unicode(255))
    content = db.Column(db.UnicodeText)
    updated_at = db.Column(db.DateTime, default=datetime.now)
    created_at = db.Column(db.DateTime, onupdate=datetime.now)
```

For more details on what the `__versioned__` property can encode, see the `SQLAlchemy-Continuum` documentation. If you have no need for the `VersioningMixin`, you can take route (2) like so:

```
class Article(db.Model):
    __versioned__ = {}
    __tablename__ = 'article'

    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    name = db.Column(db.Unicode(255))
    content = db.Column(db.UnicodeText)
```

Migrations

If you're using alembic or Flask-Migrate alongside this tool, you need to make sure a flask application context is pushed before you create new migrations. Otherwise, database fields dynamically added by the Mixins above won't be picked up by the migration tool.

If you're using alembic directly, you'll need to manually configure mappers in your app script or `create_app` factory after models are declared:

```
app = Flask(__name__)
db = SQLAlchemy(app)
continuum = Continuum(app, db)

class Article(db.Model, VersioningMixin):

    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    name = db.Column(db.Unicode(255))

continuum.configure()
```

If you're using `Flask-Migrate` to manage migrations, you don't need to manually configure the orm with versioning extensions. You can simply pass an instantiated `Flask-Migrate` plugin to `Flask-Continuum`:

```
app = Flask(__name__)
db = SQLAlchemy(app)
migrate = Migrate(app, db)
continuum = Continuum(app, db, migrate)

class Article(db.Model, VersioningMixin):

    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    name = db.Column(db.Unicode(255))
```

This will automatically configure mappers before `Flask-Migrate` performs any migration tasks.

Troubleshooting

```
>>> article = Article()
>>> db.session.add(article)
>>> db.session.commit()
...
OperationalError: no such table: transaction
```

This is usually an error caused when database tables haven't been created before a commit is made. Make sure you create database tables with `db.create_all()` before trying to commit any data to the database.

```
~$ flask db migrate
...
INFO  [alembic.env] No changes in schema detected.
```

This alembic message is produced when alembic tries to create a new database migration but doesn't detect any changes in SQLAlchemy models when trying to auto-generate the migration. It's usually caused by an application context not being pushed before migrations take place. See the *Migrations* section for information on resolving this issue.

Other Customizations

As detailed in the Overview section of the documentation, the plugin can be customized with specific triggers. The following detail what can be customized:

- `user_cls` - The name of the user table to associate with content changes.
- `current_user` - A function for returning the current user issuing a request. By default, this is determined from the `Flask-Login` plugin, but can be overwritten.
- `engine` - A SQLAlchemy engine to connect to the database. This parameter can be used if the application doesn't require the use of `Flask-SQLAlchemy`.

The code below details how you can override all of these configuration options:

```
from flask import Flask
from flask_continuum import Continuum
from sqlalchemy import create_engine

app = Flask(__name__)
engine = create_engine('postgresql://...')
```

(continues on next page)

(continued from previous page)

```
continuum = Continuum(
    engine=engine,
    user_cls='Users',
    current_user=lambda: g.user
)
continuum.init_app(app)
```

For even more in-depth information on the module and the tools it provides, see the API section of the documentation.

2.4 API

2.4.1 Base

class flask_continuum.**Continuum**(*app=None*, *db=None*, *migrate=None*, *user_cls=None*, *engine=None*, *current_user=<function fetch_current_user_id>*, *plugins=[]*)

Flask extension class for module, which sets up all flask-related capabilities provided by the module. This object can be initialized directly:

```
from flask import Flask
from flask_version import Version

app = Flask(__name__)
db = SQLAlchemy()
continuum = Continuum(app, db)
```

Or lazily via factory pattern:

```
db = SQLAlchemy()
continuum = Continuum(db=db)
app = Flask(__name__)
continuum.init_app(app)
```

To configure SQLAlchemy-Continuum with additional plugins, use the `plugins` argument to the extension:

```
from sqlalchemy.continuum.plugins import PropertyModTrackerPlugin

db = SQLAlchemy()
continuum = Continuum(db=db, plugins=[PropertyModTrackerPlugin()])
app = Flask(__name__)
continuum.init_app(app)
```

You can also use this plugin with sqlalchemy directly (i.e. not using Flask-SQLAlchemy). To do so, simply pass the database engine to this plugin upon instantiation:

```
engine = create_engine('postgresql://...')
continuum = Continuum(engine=engine)
app = Flask(__name__)
continuum.init_app(app)
```

Finally, to associate all transactions with users from a user table in the application database, you can set the *user_cls* parameter to the name of the table where users are stored:

```
app = Flask(__name__)
db = SQLAlchemy(app)
continuum = Continuum(app, db, user_cls='Users')
```

Arguments: app (Flask): Flask application to associate with plugin. db (SQLAlchemy): SQLAlchemy extension to associate with plugin. user_cls (str): Name of user class used in application. engine (Engine): SQLAlchemy engine to associate with plugin. current_user (callable): Callable object to determine user associated

with request.

plugins (list): List of other SQLAlchemy-Continuum plugins to install. See: **'https://sqlalchemy-continuum.readthedocs.io/en/latest/plugins.html'_** for more information.

init_app(*app*, *db=None*)
Initialize application via lazy factory pattern.

Args: app (Flask): Flask application. db (SQAlchemy): Flask SQLAlchemy extension.

2.4.2 Databse Mixins

class flask_continuum.**VersioningMixin**
Database mixin adding versioning support and additional helper methods to content models in application. To use this mixin in a model, you can configure it like so:

```
class Article(db.Model, VersioningMixin):
    __tablename__ = 'article'

    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    name = db.Column(db.Unicode(255))
    content = db.Column(db.UnicodeText)
    updated_at = db.Column(db.DateTime, default=datetime.now)
    created_at = db.Column(db.DateTime, onupdate=datetime.now)
```

This will implicitly add versioning support to the model.

changeset
Return SQLAlchemy-Continuum changeset for object.

modified
Return boolean describing if object has been modified.

records
Return list of records in versioning history.

Index