The Flask Mega-Tutorial Part IV: Database

December 26 2017

Posted by Miguel Grinberg under Programming , Database , Python , Flask .

Tweet   Like   Share

This is the fourth installment of the Flask Mega-Tutorial series, in which I'm going to tell you how to work with *databases*.

For your reference, below is a list of the articles in this series.

Note 1: If you are looking for the legacy version of this tutorial, it's here.

Note 2: If you would like to support my work on this blog, or just don't have patience to wait for weekly articles, I am offering the complete version of this tutorial packaged as an ebook or a set of videos. For more information, visit courses.miguelgrinberg.com.

The topic of this chapter is extremely important. For most applications, there is going to be a need to maintain persistent data that can be retrieved efficiently, and this is exactly what *databases* are made for.

The GitHub links for this chapter are: Browse, Zip, Diff.

Databases in Flask

As I'm sure you have heard already, Flask does not support databases natively. This is one of the many areas in which Flask is intentionally not opinionated, which is great, because you have the freedom to choose the database that best fits your application instead of being forced to adapt to one.

There are great choices for databases in Python, many of them with Flask extensions that make a better integration with the application. The databases can be separated into two big groups, those that follow the *relational* model, and those that do not. The latter group is often called *NoSQL*, indicating that they do not implement the popular relational query language SQL. While there are great database products in both groups, my opinion is that relational databases are a better match for applications that have structured data such as lists of users, blog posts, etc., while NoSQL databases tend to be better for data that has a less defined structure. This application, like most others, can be implemented using either type of database, but for the reasons stated above, I'm going to go with a relational database.

In Chapter 3 I showed you a first Flask extension. In this chapter I'm going to use two more. The first is Flask-SQLAlchemy, an extension that provides a Flask-friendly wrapper to the popular SQLAlchemy package, which is an Object Relational Mapper or ORM. ORMs allow applications to manage a database using high-level entities such as classes, objects and methods instead of tables and SQL. The job of the ORM is to translate the high-level operations into database commands.

The nice thing about SQLAlchemy is that it is an ORM not for one, but for many relational databases. SQLAlchemy supports a long list of database engines, including the popular MySQL, PostgreSQL and SQLite. This is extremely powerful, because you can do your development using a simple SQLite database that does not require a server, and then when

the time comes to deploy the application on a production server you can choose a more robust MySQL or PostgreSQL server, without having to change your application.

To install Flask-SQLAlchemy in your virtual environment, make sure you have activated it first, and then run:

```
(venv) $ pip install flask-sqlalchemy
```

Database Migrations

Most database tutorials I've seen cover creation and use of a database, but do not adequately address the problem of making updates to an existing database as the application needs change or grow. This is hard because relational databases are centered around structured data, so when the structure changes the data that is already in the database needs to be *migrated* to the modified structure.

The second extension that I'm going to present in this chapter is Flask-Migrate, which is actually one created by yours truly. This extension is a Flask wrapper for Alembic, a database migration framework for SQLAlchemy. Working with database migrations adds a bit of work to get a database started, but that is a small price to pay for a robust way to make changes to your database in the future.

The installation process for Flask-Migrate is similar to other extensions you have seen:

```
(venv) $ pip install flask-migrate
```

Flask-SQLAlchemy Configuration

During development, I'm going to use a SQLite database. SQLite databases are the most convenient choice for developing small applications, sometimes even not so small ones, as each database is stored in a single file on disk and there is no need to run a database server like MySQL and PostgreSQL.

We have two new configuration items to add to the config file:

config.py: Flask-SQLAlchemy configuration

```python
import os
basedir = os.path.abspath(os.path.dirname(__file__))

class Config(object):
    # ...
    SQLALCHEMY_DATABASE_URI = os.environ.get('DATABASE_URL') or \
        'sqlite:///' + os.path.join(basedir, 'app.db')
    SQLALCHEMY_TRACK_MODIFICATIONS = False
```

The Flask-SQLAlchemy extension takes the location of the application's database from the `SQLALCHEMY_DATABASE_URI` configuration variable. As you recall from Chapter 3, it is in general a good practice to set configuration from environment variables, and provide a fallback value when the environment does not define the variable. In this case I'm taking the database URL from the `DATABASE_URL` environment variable, and if that isn't defined, I'm configuring a database named *app.db* located in the main directory of the application, which is stored in the `basedir` variable.

The `SQLALCHEMY_TRACK_MODIFICATIONS` configuration option is set to `False` to disable a feature of Flask-SQLAlchemy that I do not need, which is to signal the application every time a

change is about to be made in the database.

The database is going to be represented in the application by the *database instance*. The database migration engine will also have an instance. These are objects that need to be created after the application, in the *app/__init__.py* file:

app/__init__.py: Flask-SQLAlchemy and Flask-Migrate initialization

```python
from flask import Flask
from config import Config
from flask_sqlalchemy import SQLAlchemy
from flask_migrate import Migrate

app = Flask(__name__)
app.config.from_object(Config)
db = SQLAlchemy(app)
migrate = Migrate(app, db)

from app import routes, models
```

I have made three changes to the init script. First, I have added a `db` object that represents the database. Then I have added another object that represents the migration engine. Hopefully you see a pattern in how to work with Flask extensions. Most extensions are initialized as these two. Finally, I'm importing a new module called `models` at the bottom. This module will define the structure of the database.

Database Models

The data that will be stored in the database will be represented by a collection of classes, usually called *database models*. The ORM layer within SQLAlchemy will do the translations required to map objects created from these classes into rows in the proper database tables.

Let's start by creating a model that represents users. Using the WWW SQL Designer tool, I have made the following diagram to represent the data that we want to use in the users table:
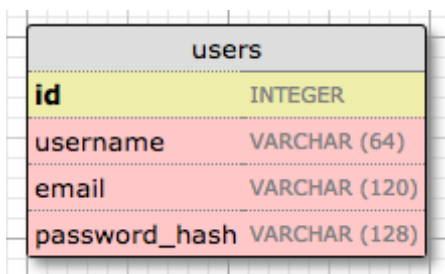
The `id` field is usually in all models, and is used as the *primary key*. Each user in the database will be assigned a unique id value, stored in this field. Primary keys are, in most cases, automatically assigned by the database, so I just need to provide the `id` field marked as a primary key.

The `username`, `email` and `password_hash` fields are defined as strings (or `VARCHAR` in database jargon), and their maximum lengths are specified so that the database can optimize space usage. While the `username` and `email` fields are self-explanatory, the `password_hash` fields deserves some attention. I want to make sure the application that I'm building adopts security best practices, and for that reason I will not be storing user passwords in the database. The problem with storing passwords is that if the database ever becomes compromised, the attackers will have access to the passwords, and that could be devastating

for users. Instead of writing the passwords directly, I'm going to write *password hashes*, which greatly improve security. This is going to be the topic of another chapter, so don't worry about it too much for now.

So now that I know what I want for my users table, I can translate that into code in the new *app/models.py* module:

app/models.py: User database model

```python
from app import db

class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(64), index=True, unique=True)
    email = db.Column(db.String(120), index=True, unique=True)
    password_hash = db.Column(db.String(128))

    def __repr__(self):
        return '<User {}>'.format(self.username)
```

The `User` class created above inherits from `db.Model`, a base class for all models from Flask-SQLAlchemy. This class defines several fields as class variables. Fields are created as instances of the `db.Column` class, which takes the field type as an argument, plus other optional arguments that, for example, allow me to indicate which fields are unique and indexed, which is important so that database searches are efficient.

The `__repr__` method tells Python how to print objects of this class, which is going to be useful for debugging. You can see the `__repr__()` method in action in the Python interpreter session below:

```
>>> from app.models import User
>>> u = User(username='susan', email='susan@example.com')
>>> u
<User susan>
```

Creating The Migration Repository

The model class created in the previous section defines the initial database structure (or *schema*) for this application. But as the application continues to grow, there is going to be a need change that structure, very likely to add new things, but sometimes also to modify or remove items. Alembic (the migration framework used by Flask-Migrate) will make these schema changes in a way that does not require the database to be recreated from scratch.

To accomplish this seemingly difficult task, Alembic maintains a *migration repository*, which is a directory in which it stores its migration scripts. Each time a change is made to the database schema, a migration script is added to the repository with the details of the change. To apply the migrations to a database, these migration scripts are executed in the sequence they were created.

Flask-Migrate exposes its commands through the `flask` command. You have already seen `flask run`, which is a sub-command that is native to Flask. The `flask db` sub-command is added by Flask-Migrate to manage everything related to database migrations. So let's create the migration repository for microblog by running `flask db init`:

```
(venv) $ flask db init
  Creating directory /home/miguel/microblog/migrations ... done
```

```
Creating directory /home/miguel/microblog/migrations/versions ... done
Generating /home/miguel/microblog/migrations/alembic.ini ... done
Generating /home/miguel/microblog/migrations/env.py ... done
Generating /home/miguel/microblog/migrations/README ... done
Generating /home/miguel/microblog/migrations/script.py.mako ... done
Please edit configuration/connection/logging settings in
'/home/miguel/microblog/migrations/alembic.ini' before proceeding.
```

Remember that the `flask` command relies on the `FLASK_APP` environment variable to know where the Flask application lives. For this application, you want to set `FLASK_APP=microblog.py`, as discussed in Chapter 1.

After you run this command, you will find a new *migrations* directory, with a few files and a *versions* sub-directory inside. All these files should be treated as part of your project from now on, and in particular, should be added to source control.

The First Database Migration

With the migration repository in place, it is time to create the first database migration, which will include the users table that maps to the `User` database model. There are two ways to create a database migration: manually or automatically. To generate a migration automatically, Alembic compares the database schema as defined by the database models, against the actual database schema currently used in the database. It then populates the migration script with the changes necessary to make the database schema match the application models. In this case, since there is no previous database, the automatic migration will add the entire `User` model to the migration script. The `flask db migrate` sub-command generates these automatic migrations:

```
(venv) $ flask db migrate -m "users table"
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.autogenerate.compare] Detected added table 'user'
INFO  [alembic.autogenerate.compare] Detected added index 'ix_user_email' on '['email']'
INFO  [alembic.autogenerate.compare] Detected added index 'ix_user_username' on '['username']'
  Generating /home/miguel/microblog/migrations/versions/e517276bb1c2_users_table.py ... done
```

The output of the command gives you an idea of what Alembic included in the migration. The first two lines are informational and can usually be ignored. It then says that it found a user table and two indexes. Then it tells you where it wrote the migration script. The `e517276bb1c2` code is an automatically generated unique code for the migration (it will be different for you). The comment given with the `-m` option is optional, it adds a short descriptive text to the migration.

The generated migration script is now part of your project, and needs to be incorporated to source control. You are welcome to inspect the script if you are curious to see how it looks. You will find that it has two functions called `upgrade()` and `downgrade()`. The `upgrade()` function applies the migration, and the `downgrade()` function removes it. This allows Alembic to migrate the database to any point in the history, even to older versions, by using the downgrade path.

The `flask db migrate` command does not make any changes to the database, it just generates the migration script. To apply the changes to the database, the `flask db upgrade` command must be used.

```
(venv) $ flask db upgrade
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
```

```
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.runtime.migration] Running upgrade  -> e517276bb1c2, users table
```

Because this application uses SQLite, the `upgrade` command will detect that a database does not exist and will create it (you will notice a file named *app.db* is added after this command finishes, that is the SQLite database). When working with database servers such as MySQL and PostgreSQL, you have to create the database in the database server before running `upgrade`.

Note that Flask-SQLAlchemy uses a "snake case" naming convention for database tables by default. For the `User` model above, the corresponding table in the database will be named `user`. For a `AddressAndPhone` model class, the table would be named `address_and_phone`. If you prefer to choose your own table names, you can add an attribute named `__tablename__` to the model class, set to the desired name as a string.

Database Upgrade and Downgrade Workflow

The application is in its infancy at this point, but it does not hurt to discuss what is going to be the database migration strategy going forward. Imagine that you have your application on your development machine, and also have a copy deployed to a production server that is online and in use.

Let's say that for the next release of your app you have to introduce a change to your models, for example a new table needs to be added. Without migrations you would need to figure out how to change the schema of your database, both in your development machine and then again in your server, and this could be a lot of work.

But with database migration support, after you modify the models in your application you generate a new migration script (`flask db migrate`), you probably review it to make sure the automatic generation did the right thing, and then apply the changes to your development database (`flask db upgrade`). You will add the migration script to source control and commit it.

When you are ready to release the new version of the application to your production server, all you need to do is grab the updated version of your application, which will include the new migration script, and run `flask db upgrade`. Alembic will detect that the production database is not updated to the latest revision of the schema, and run all the new migration scripts that were created after the previous release.

As I mentioned earlier, you also have a `flask db downgrade` command, which undoes the last migration. While you will be unlikely to need this option on a production system, you may find it very useful during development. You may have generated a migration script and applied it, only to find that the changes that you made are not exactly what you need. In this case, you can downgrade the database, delete the migration script, and then generate a new one to replace it.

Database Relationships

Relational databases are good at storing relations between data items. Consider the case of a user writing a blog post. The user will have a record in the `users` table, and the post will have a record in the `posts` table. The most efficient way to record who wrote a given post is to link the two related records.

Once a link between a user and a post is established, the database can answer queries about this link. The most trivial one is when you have a blog post and need to know what user wrote it. A more complex query is the reverse of this one. If you have a user, you may want to know all the posts that this user wrote. Flask-SQLAlchemy will help with both types of queries.

Let's expand the database to store blog posts to see relationships in action. Here is the schema for a new `posts` table:
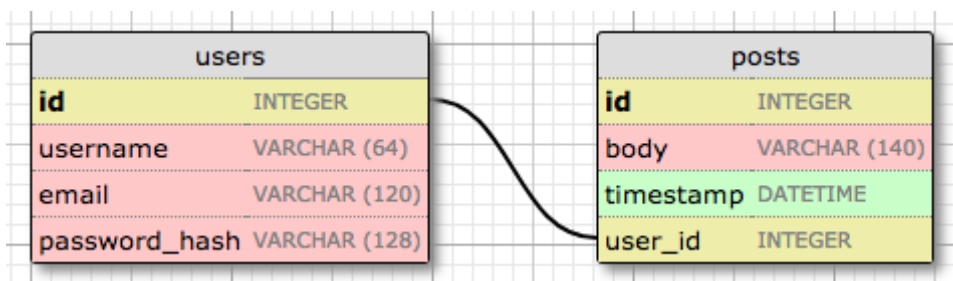
The `posts` table will have the required `id`, the `body` of the post and a `timestamp`. But in addition to these expected fields, I'm adding a `user_id` field, which links the post to its author. You've seen that all users have a `id` primary key, which is unique. The way to link a blog post to the user that authored it is to add a reference to the user's `id`, and that is exactly what the `user_id` field is. This `user_id` field is called a *foreign key*. The database diagram above shows foreign keys as a link between the field and the `id` field of the table it refers to. This kind of relationship is called a *one-to-many*, because "one" user writes "many" posts.

The modified *app/models.py* is shown below:

app/models.py: Posts database table and relationship

```python
from datetime import datetime
from app import db


class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(64), index=True, unique=True)
    email = db.Column(db.String(120), index=True, unique=True)
    password_hash = db.Column(db.String(128))
    posts = db.relationship('Post', backref='author', lazy='dynamic')

    def __repr__(self):
        return '<User {}>'.format(self.username)

class Post(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    body = db.Column(db.String(140))
    timestamp = db.Column(db.DateTime, index=True, default=datetime.utcnow)
    user_id = db.Column(db.Integer, db.ForeignKey('user.id'))

    def __repr__(self):
        return '<Post {}>'.format(self.body)
```

The new `Post` class will represent blog posts written by users. The `timestamp` field is going to be indexed, which is useful if you want to retrieve posts in chronological order. I have also added a `default` argument, and passed the `datetime.utcnow` function. When you pass a function as a default, SQLAlchemy will set the field to the value of calling that function (note that I did not include the `()` after `utcnow`, so I'm passing the function itself, and not the result of calling it). In general, you will want to work with UTC dates and times in a server application. This ensures that you are using uniform timestamps regardless of where the users are located. These timestamps will be converted to the user's local time when they are displayed.

The `user_id` field was initialized as a foreign key to `user.id`, which means that it references an `id` value from the users table. In this reference the `user` part is the name of the database table for the model. It is an unfortunate inconsistency that in some instances such as in a `db.relationship()` call, the model is referenced by the model class, which typically starts with an uppercase character, while in other cases such as this `db.ForeignKey()` declaration, a model is given by its database table name, for which SQLAlchemy automatically uses lowercase characters and, for multi-word model names, snake case.

The `User` class has a new `posts` field, that is initialized with `db.relationship`. This is not an actual database field, but a high-level view of the relationship between users and posts, and for that reason it isn't in the database diagram. For a one-to-many relationship, a `db.relationship` field is normally defined on the "one" side, and is used as a convenient way to get access to the "many". So for example, if I have a user stored in `u`, the expression `u.posts` will run a database query that returns all the posts written by that user. The first argument to `db.relationship` is the model class that represents the "many" side of the relationship. This argument can be provided as a string with the class name if the model is defined later in the module. The `backref` argument defines the name of a field that will be added to the objects of the "many" class that points back at the "one" object. This will add a `post.author` expression that will return the user given a post. The `lazy` argument defines how the database query for the relationship will be issued, which is something that I will discuss later. Don't worry if these details don't make much sense just yet, I'll show you examples of this at the end of this article.

Since I have updates to the application models, a new database migration needs to be generated:

```
(venv) $ flask db migrate -m "posts table"
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.autogenerate.compare] Detected added table 'post'
INFO  [alembic.autogenerate.compare] Detected added index 'ix_post_timestamp' on '['timestamp
']'
  Generating /home/miguel/microblog/migrations/versions/780739b227a7_posts_table.py ... done
```

And the migration needs to be applied to the database:

```
(venv) $ flask db upgrade
INFO  [alembic.runtime.migration] Context impl SQLiteImpl.
INFO  [alembic.runtime.migration] Will assume non-transactional DDL.
INFO  [alembic.runtime.migration] Running upgrade e517276bb1c2 -> 780739b227a7, posts table
```

If you are storing your project in source control, also remember to add the new migration script
to it.

Play Time

I have made you suffer through a long process to define the database, but I haven't shown
you how everything works yet. Since the application does not have any database logic yet,
let's play with the database in the Python interpreter to familiarize with it. So go ahead and fire
up Python by running `python`. Make sure your virtual environment is activated before you
start the interpreter.

Once in the Python prompt, let's import the database instance and the models:

```
>>> from app import db
>>> from app.models import User, Post
```

Start by creating a new user:

```
>>> u = User(username='john', email='john@example.com')
>>> db.session.add(u)
>>> db.session.commit()
```

Changes to a database are done in the context of a session, which can be accessed as `db.session`. Multiple changes can be accumulated in a session and once all the changes have been registered you can issue a single `db.session.commit()`, which writes all the changes atomically. If at any time while working on a session there is an error, a call to `db.session.rollback()` will abort the session and remove any changes stored in it. The important thing to remember is that changes are only written to the database when `db.session.commit()` is called. Sessions guarantee that the database will never be left in an inconsistent state.

Let's add another user:

```
>>> u = User(username='susan', email='susan@example.com')
>>> db.session.add(u)
>>> db.session.commit()
```

The database can answer a query that returns all the users:

```
>>> users = User.query.all()
>>> users
```

```
[<User john>, <User susan>]
>>> for u in users:
...     print(u.id, u.username)
...
1 john
2 susan
```

All models have a `query` attribute that is the entry point to run database queries. The most basic query is that one that returns all elements of that class, which is appropriately named `all()`. Note that the `id` fields were automatically set to 1 and 2 when those users were added.

Here is another way to do queries. If you know the `id` of a user, you can retrieve that user as follows:

```
>>> u = User.query.get(1)
>>> u
<User john>
```

Now let's add a blog post:

```
>>> u = User.query.get(1)
>>> p = Post(body='my first post!', author=u)
>>> db.session.add(p)
>>> db.session.commit()
```

I did not need to set a value for the `timestamp` field because that field has a default, which you can see in the model definition. And what about the `user_id` field? Recall that the `db.relationship` that I created in the `User` class adds a `posts` attribute to users, and also

a `author` attribute to posts. I assign an author to a post using the `author` virtual field instead of having to deal with user IDs. SQLAlchemy is great in that respect, as it provides a high-level abstraction over relationships and foreign keys.

To complete this session, let's look at a few more database queries:

```
>>> # get all posts written by a user
>>> u = User.query.get(1)
>>> u
<User john>
>>> posts = u.posts.all()
>>> posts
[<Post my first post!>]

>>> # same, but with a user that has no posts
>>> u = User.query.get(2)
>>> u
<User susan>
>>> u.posts.all()
[]

>>> # print post author and body for all posts
>>> posts = Post.query.all()
>>> for p in posts:
...     print(p.id, p.author.username, p.body)
...
1 john my first post!

# get all users in reverse alphabetical order
>>> User.query.order_by(User.username.desc()).all()
[<User susan>, <User john>]
```

The [Flask-SQLAlchemy](#) documentation is the best place to learn about the many options that are available to query the database.

To complete this section, let's erase the test users and posts created above, so that the database is clean and ready for the next chapter:

```
>>> users = User.query.all()
>>> for u in users:
...     db.session.delete(u)
...
>>> posts = Post.query.all()
>>> for p in posts:
...     db.session.delete(p)
...
>>> db.session.commit()
```

Shell Context

Remember what you did at the start of the previous section, right after starting a Python interpreter? The first thing you did was to run some imports:

```
>>> from app import db
>>> from app.models import User, Post
```

While you work on your application, you will need to test things out in a Python shell very often, so having to repeat the above imports every time is going to get tedious. The `flask shell` command is another very useful tool in the `flask` umbrella of commands. The `shell`

command is the second "core" command implemented by Flask, after `run` . The purpose of this command is to start a Python interpreter in the context of the application. What does that mean? See the following example:

```
(venv) $ python
>>> app
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'app' is not defined
>>>

(venv) $ flask shell
>>> app
<Flask 'app'>
```

With a regular interpreter session, the `app` symbol is not known unless it is explicitly imported, but when using `flask shell` , the command pre-imports the application instance. The nice thing about `flask shell` is not that it pre-imports `app` , but that you can configure a "shell context", which is a list of other symbols to pre-import.

The following function in *microblog.py* creates a shell context that adds the database instance and models to the shell session:

```
from app import app, db
from app.models import User, Post

@app.shell_context_processor
def make_shell_context():
    return {'db': db, 'User': User, 'Post': Post}
```

The `app.shell_context_processor` decorator registers the function as a shell context function. When the `flask shell` command runs, it will invoke this function and register the items returned by it in the shell session. The reason the function returns a dictionary and not a list is that for each item you have to also provide a name under which it will be referenced in the shell, which is given by the dictionary keys.

After you add the shell context processor function you can work with database entities without having to import them:

```
(venv) $ flask shell
>>> db
<SQLAlchemy engine=sqlite:////Users/migu7781/Documents/dev/flask/microblog2/app.db>
>>> User
<class 'app.models.User'>
>>> Post
<class 'app.models.Post'>
```

If you try the above and get `NameError` exceptions when you access `db`, `User` and `Post`, then the `make_shell_context()` function is not being registered with Flask. The most likely cause of this is that you have not set `FLASK_APP=microblog.py` in the environment. In that case, go back to Chapter 1 and review how to set the `FLASK_APP` environment variable. If you often forget to set this variable when you open new terminal windows, you may consider adding a *.flaskenv* file to your project, as described at the end of that chapter.

Hello, and thank you for visiting my blog! If you enjoyed this article, please consider supporting my work on this blog on Patreon!

Tweet   Like   Share

543 comments

#1 Polow said 2 years ago

This is really helpful. Thanks a lot!

#2 Patrick Kennedy said 2 years ago

Hi Miguel,
This was a great blog post and I especially like setting up the database migration steps from the beginning of a project.

It's defined by the .gitignore file, but the app.db file (when using SQLite) should not be added to the source control of this project. I almost added app.db by accident (I had my own version of .gitignore) while going through this blog post, so just wanted to mention this for others to avoid.

Thank you,
Patrick

#3 Kevin said 2 years ago

Great stuff so far. Looking forward to the next articles coming out in January.

Thanks!

#4 **Francis** said 2 years ago

Hi Miguel,

Thanks as always for the great tutorials, bought your ebook btw. Why is it that in the 'user_id' field in the 'Post' model the ForeignKey argument is 'user.id' instead of 'User.id'? Where does the user object come from at this point? Please give some pointers, thanks and this is probably not the last of my questions.
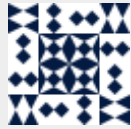
#5 **Jamie** said 2 years ago

Hi Miguel, Thought I'd leave a note to say THANK YOU! You write very succinctly and I like your style. As a completely new starter in Python I'm learning a lot. Can't wait for the next one.

Cheers,

Jamie

#6 **Karen** said 2 years ago

I keep getting error in pycharm Python console

sqlalchemy.exc.ProgrammingError: (sqlite3.ProgrammingError) SQLite objects created in a thread can only be used in that same thread.The object was created in thread id 123145332412416 and this is thread id 140735907148608 [SQL: 'SELECT user.id AS user_id, user.username AS user_username, user.email AS user_email, user.password_hash AS user_password_hash \nFROM user'] [parameters: [{}]]
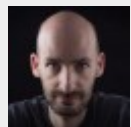
#7 **Miguel Grinberg** said 2 years ago

@Francis: the "user" part refers to the table in the database, which is named as the class name, but all in lowercase.

#8 **Miguel Grinberg** said 2 years ago

@Karen: Hmm, not sure about this one. Do you get the error when you run in the standalone Python console, as shown in this tutorial? Maybe PyCharm has a way to run the console that conflicts with sqlite.

#9 **fisher** said 2 years ago

Hey Miguel,
I'm following along and I've got an issue with adding a test post. My shell is saying that author is an invalid kward for the Post class, so I'm guessing for some reason my db hasn't created the virtual field?
Pastebin of Stack Trace: https://pastebin.com/QstTYQ0L
Happy Holidays!
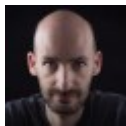
#10 **Miguel Grinberg** said 2 years ago

@fisher: My guess is that you did not define the relationship. Did you add the "posts = db.relationship(...)" line in your User model?
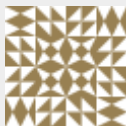
#11 **Ben** said 2 years ago

Awesome stuff Miguel!

I was wondering if you cover implementing many to many relationships in this tutorial.

I'm trying to do that now and its proving to be a big headache it terms of making sure there are no

duplicate associations and deleting associations.

Thanks!

#12 **majid** said 2 years ago

Hey Miguel,
can you say that how Making links to the next and previous pages too home page?

Thanks!
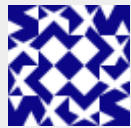
#13 **fisher** said 2 years ago

@miguel: Yep, that was it. My mistake trying to skim large paragraphs rather than paying attention!
Thanks!
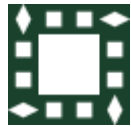
#14 **Jose Luis** said 2 years ago

It is a great idea to refresh this tutorial. Keep on the good work!

I have a question regarding the possibility of loading sqlite extensions in order to expand its functionality.

In my case I am trying to use spatialite, a module that adds spatial functions to the DB; with sqlalchemy, this is done defining an event that listens for the DB connections and associating a function that perfrms the load of the extension:

engine = create_engine('sqlite:///devdata.db', module=sqlite, echo=False)

this enables the extension on each connection
@event.listens_for(engine, "connect")

```
def connect(dbapi_connection, connection_rec):
    dbapi_connection.enable_load_extension(True)
    dbapi_connection.execute("SELECT load_extension('mod_spatialite')")
```

Is it possible to do this with Flask-sqlalchemy?

Happy new year to alll!

#15 **Miguel Grinberg** said 2 years ago

Ben: Many-to-many relationships are coming later in the tutorial, yes.

#16 **Miguel Grinberg** said 2 years ago

@majid: there is going to be a chapter dedicated to Pagination.
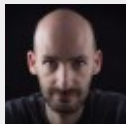
#17 **Miguel Grinberg** said 2 years ago

@Jose Luis: never done this myself, but it is likely possible if you subclass the SQAlchemy class and provide some additional logic in the apply_driver_hacks() method (see https://github.com/mitsuhiko/flask-sqlalchemy/blob/master/flask_sqlalchemy/__init__.py#L820).

#18 **Dawn** said 2 years ago

Thanks for the tutorial Miguel!
I'm working on Chapter 4. Everything worked at listing 4.3 - then I began the migration module. I had a 2 typos in models.py. I fixed the errors, ran the migration/upgrade and the app threw off errors in files I didn't know I had on the system....dozens and dozens of errors. Then I remembered the downgrade command. Worked perfectly!! I'm still not sure what I did wrong, but I was able to back

out of the last db upgrade and get back on track with your tutorial. This is the kind of error that brought me to a complete stop in other tutorials - yours are great! Wish me luck as I continue
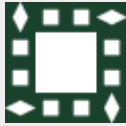
For the record this is how I managed to make it work, I included the load command in the _init_.py file:

```
from flask import Flask
from config import Config
from flask_sqlalchemy import SQLAlchemy, event

db = SQLAlchemy()

def create_app():
    global db
    app = Flask(__name__)

    app.config....
    db.init_app(app)

    with app.app_context():
        @event.listens_for(db.engine, "first_connect")
        def connect(sqlite, connection_rec):
            sqlite.enable_load_extension(True)
            sqlite.execute("SELECT load_extension('C:\\spatialite32\\mod_spatialite')")
            sqlite.enable_load_extension(False)

        @event.listens_for(db.engine, "connect")
        def connect(sqlite, connection_rec):
            sqlite.enable_load_extension(True)
            sqlite.execute("SELECT load_extension('C:\\spatialite32\\mod_spatialite')")
            sqlite.enable_load_extension(False)
```

```
        return app

    app = create_app()

    from app import routes, models
```

Hi Miguel - thanks for your tutorial! I bought the e-book and couldn't be happier. Real bargain for $15.

Question regarding...
Chapter 4 - Databases - Setting environmental variable - SQLALCHEMY_DATABASE_URI - Windows 10

Do you know the Windows 10 Powershell command to set the environmental variable? I'm doing....
(venv) ... > $env:SQLALCHEMY_DATABASE_URI="sqlite:///C:\Projects\Microblog\app.db"

But then when I do "flask db migrate", I get a UserWarning that says the above variable isn't set.

Any suggestions?

#23 **Miguel Grinberg** said 2 years ago

@Michael: I don't use Powershell myself, but the environment variable for the database is called DATABASE_URL. In general I use the same name for the configuration and the environment variable, this one is an exception.

#24 **James** said 2 years ago

Forgive in advance if my question is silly. At what point does the code for this section show that it is using SQLite database? From my reading SQLAlchemy can be used on any database, but I do not explicitly find where it is linked to SQLite.
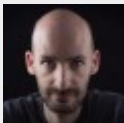
#25 **Miguel Grinberg** said 2 years ago

@James: See the "Flask-SQLAlchemy Configuration" section, near the beginning of the article.

Leave a Comment

Name

Email

Comment

Captcha

[ ] I'm not a robot

reCAPTCHA
Privacy - Terms

Submit

MicroPython and the Internet of Things

If you would you like to support my MicroPython tutorial series on this blog and as a reward have access to the complete tutorial nicely structured as an ebook and 6+ hours of video material, you can now order it from my Courses site.

About Miguel

Welcome to my blog!

I'm a software engineer, photographer and filmmaker, currently living in Drogheda, Ireland.

You can also find me on Facebook, Google+, LinkedIn, Github and Twitter.

Thank you for visiting!

Categories

AWS (1)