The New and Improved Flask Mega-Tutorial

(2024 Edition)

Miguel Grinberg

The New and Improved Flask Mega-Tutorial (2024 Edition)

Miguel Grinberg

Jan 09, 2025

4. Database

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[1]*, Zip*[2]*, Diff*[3]*.*

4.1. 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[4]. 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[5], an extension that provides a Flask-friendly wrapper to the popular SQLAlchemy[6] package, which is an Object Relational Mapper[7] or ORM. ORMs allow applications

[1] https://github.com/miguelgrinberg/microblog/tree/v0.4
[2] https://github.com/miguelgrinberg/microblog/archive/v0.4.zip
[3] https://github.com/miguelgrinberg/microblog/compare/v0.3...v0.4
[4] https://en.wikipedia.org/wiki/SQL
[5] http://packages.python.org/Flask-SQLAlchemy
[6] http://www.sqlalchemy.org
[7] http://en.wikipedia.org/wiki/Object-relational_mapping

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[8], PostgreSQL[9] and SQLite[10]. 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
```

4.2. 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[11], which is actually one created by myself. This extension is a Flask wrapper for Alembic[12], 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:

[8] https://www.mysql.com/
[9] https://www.postgresql.org/
[10] https://www.sqlite.org/
[11] https://github.com/miguelgrinberg/flask-migrate
[12] https://alembic.sqlalchemy.org/

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

4.3. 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.

Flask-SQLAlchemy needs a new configuration item added to the config file:

Listing 4.1: *config.py*: Flask-SQLAlchemy configuration

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

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

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 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:

Listing 4.2: *app/__init__.py*: Flask-SQLAlchemy and Flask-Migrate initialization

```
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)
```

(continues on next page)

(continued from previous page)

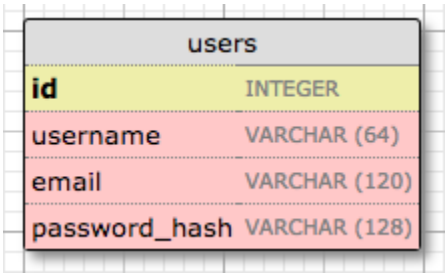
```
migrate = Migrate(app, db)

from app import routes, models
```

I have made three changes to the *__init__.py* file. First, I have added a `db` object that represents the database. Then I added `migrate`, to represent the database migration engine. Hopefully you see a pattern in how to work with Flask extensions. Most extensions are initialized as these two. In the last change, I'm importing a new module called `models` at the bottom. This module will define the structure of the database.

4.4. 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[13] 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 that the application that I'm building adopts security best practices, and for that reason I will not be storing user passwords in plain text. 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.

[13] http://ondras.zarovi.cz/sql/demo

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:

Listing 4.3: *app/models.py*: User database model

```
from typing import Optional
import sqlalchemy as sa
import sqlalchemy.orm as so
from app import db

class User(db.Model):
    id: so.Mapped[int] = so.mapped_column(primary_key=True)
    username: so.Mapped[str] = so.mapped_column(sa.String(64), index=True,
                                                unique=True)
    email: so.Mapped[str] = so.mapped_column(sa.String(120), index=True,
                                             unique=True)
    password_hash: so.Mapped[Optional[str]] = so.mapped_column(sa.String(256))

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

I start by importing the `sqlalchemy` and `sqlalchemy.orm` modules from the SQLAlchemy package, which provide most of the elements that are needed to work with a database. The `sqlalchemy` module includes general purpose database functions and classes such as types and query building helpers, while `sqlalchemy.orm` provides the support for using models. Given that these two module names are long and will need to be referenced often, the `sa` and `so` aliases are defined directly in the import statements. The `db` instance from Flask-SQLAlchemy and the `Optional` typing hint from Python are imported as well.

The `User` class created above will represent users stored in the database. The class inherits from `db.Model`, a base class for all models from Flask-SQLAlchemy. The `User` model defines several fields as class variables. These are the columns that will be created in the corresponding database table.

Fields are assigned a type using Python *type hints*, wrapped with SQLAlchemy's `so.Mapped` generic type. A type declaration such as `so.Mapped[int]` or `so.Mapped[str]` define the type of the column, and also make values required, or *non-nullable* in database terms. To define a column that is allowed to be empty or *nullable*, the `Optional` helper from Python is also added, as `password_hash` demonstrates.

In most cases defining a table column requires more than the column type. SQLAlchemy uses a `so.mapped_column()` function call assigned to each column to provide this additional configuration. In the case of `id` above, the column is configured as the primary key. For string columns many databases require a length to be given, so this is also included. I have included other optional arguments that allow me to indicate which fields are unique and indexed, which is important so that database is consistent and 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>
```

4.5. 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, it is likely that I will need to make changes to that structure such as adding new things, and sometimes 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 every time a change is made.

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` to the value `microblog.py`, as discussed in *Chapter 1*. If you included a *.flaskenv* file in your project, then the all sub-commands of the `flask` command will automatically have access to the application.

After you run the `flask db init` command, you will find a new *migrations* directory, with a few files and a *versions* subdirectory inside. All these files should be treated as part of your project from now on, and in particular, should be added to source control along with your application code.

4.6. 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` value is an automatically generated and unique code for the migration (it will be different for you). The comment given with the `-m` option is optional, it just adds a short descriptive text to the migration.

The generated migration script is now part of your project, and if you are using git or other source control tool, it needs to be incorporated as an additional source file, along with all other files stored in the *migrations* directory. 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.

4.7. 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 application 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 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.

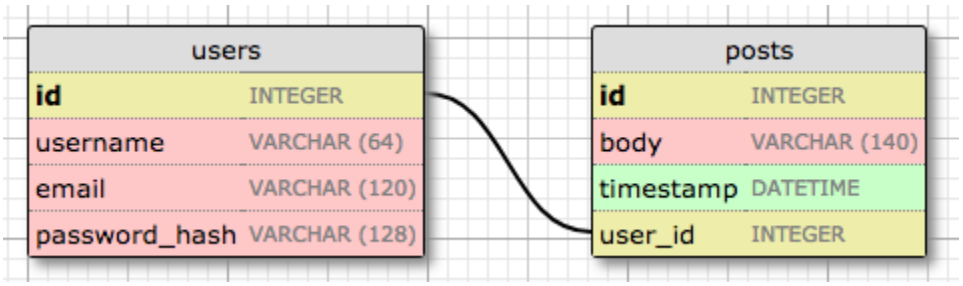
As I mentioned earlier, you also have a `flask db downgrade` command, which undoes the last migration. While you will be unlikely to need this option on a production system, you may find it very useful during development. You may have generated a migration script and applied it, only to find that the 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.

4.8. 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. SQLAlchemy helps 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 for. This `user_id` field is called a *foreign key*, because it references a primary key of another table. 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:

Listing 4.4: *app/models.py*: Posts database table and relationship

```
from datetime import datetime, timezone
from typing import Optional
import sqlalchemy as sa
import sqlalchemy.orm as so
from app import db

class User(db.Model):
    id: so.Mapped[int] = so.mapped_column(primary_key=True)
    username: so.Mapped[str] = so.mapped_column(sa.String(64), index=True,
                                                unique=True)
    email: so.Mapped[str] = so.mapped_column(sa.String(120), index=True,
                                             unique=True)
    password_hash: so.Mapped[Optional[str]] = so.mapped_column(sa.String(256))

    posts: so.WriteOnlyMapped['Post'] = so.relationship(
        back_populates='author')

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

class Post(db.Model):
    id: so.Mapped[int] = so.mapped_column(primary_key=True)
    body: so.Mapped[str] = so.mapped_column(sa.String(140))
    timestamp: so.Mapped[datetime] = so.mapped_column(
        index=True, default=lambda: datetime.now(timezone.utc))
    user_id: so.Mapped[int] = so.mapped_column(sa.ForeignKey(User.id),
                                               index=True)

    author: so.Mapped[User] = so.relationship(back_populates='posts')
```

(continues on next page)