Previous Next

Managing database migrations using Flask-Migrate

Estimated Time: 1-2 hours

In this assignment you're going to add one small new feature to our blog: the ability to link an entry with its author. In order to do this you are going to create a relationship between the `User` object and the `Entry` object.

This sounds fairly straightforward right? You add a foreign key to the `Entry`, a relationship to the `User` and you're good to go. Well, not quite. Remember that you already have a load of information stored in the database which you don't want to lose by recreating the entire schema. So how do you transition between the old database models and the new one?

The answer is by using *migrations*. Migrations allow you to set up a series of scripts which allow us to easily move between different database schemas, adding and removing columns as necessary. SQLAlchemy has a migration tool called Alembic which you are going to use via Flask-Migrate, a thin wrapper which integrates nicely with our management script.

Initializing migrations

The first step is to add the migration management commands to your *manage.py* script:

```
from flask_migrate import Migrate, MigrateCommand
from blog.database import Base

class DB(object):
    def __init__(self, metadata):
        self.metadata = metadata

migrate = Migrate(app, DB(Base.metadata))
manager.add_command('db', MigrateCommand)
```

Here you create a new class called `DB` which is designed to hold your metadata object. Alembic uses the metadata to work out what the changes to the database schema should be. You then create an instance of Flask-Migrate's `Migrate` class, passing in the app and an instance of the `DB` class. Finally you use the `add_command` to add all of the commands held in the `Migrate` class to the management script.

Now try using your script to initialize the migrations. At the terminal, run `python manage.py db init`. This will create a folder called *migrations* which will store the migration scripts and the configuration for Alembic. Note that alembic will ask you to edit some configuration settings. Flask-Migrate takes care of this automatically, so you can safely ignore this message.

Changing the models

Now let's see how to run a migration on our database. First make the changes to the models:

```
from sqlalchemy import ForeignKey
from sqlalchemy.orm import relationship

class User(Base, UserMixin):
    __tablename__ = "users"

    id = Column(Integer, primary_key=True)
    name = Column(String(128))
```

```
    email = Column(String(128), unique=True)
    password = Column(String(128))
    entries = relationship("Entry", backref="author")

class Entry(Base):
    __tablename__ = "entries"

    id = Column(Integer, primary_key=True)
    title = Column(String(1024))
    content = Column(Text)
    datetime = Column(DateTime, default=datetime.datetime.now)
    author_id = Column(Integer, ForeignKey('users.id'))
```

Here you've added a one-to-many relationship between the `User` model and the `Entry` model. Now let's see how to use migrations to reflect this change in our database.

First you need to *create* a migration script. Run `python manage.py db migrate` from the command line. You should see that a new file has been created in the *migrations/versions* folder. This will contain two functions, one for upgrading the database and one for downgrading it. The upgrade function should contain a line which adds the foreign key to the `entries` table, and the downgrade function should remove the foreign key.

In this case the changes to the database have been automatically calculated by Alembic. The system for detecting changes is not perfect however, and occasionally you may find that you need to modify the migration script before you apply the changes.

Run the migration to actually *apply* the changes to the database by saying `python manage.py db upgrade`. This will run the upgrade function, adding the column to the table. Now you should be able to update the views and templates to use the author attribute of the `Entry` model. In *views.py* you can update the `add_entry_post` function to assign the logged in user as the author of an entry:

```
from flask_login import current_user

@app.route("/entry/add", methods=["POST"])
@login_required
def add_entry_post():
    entry = Entry(
        title=request.form["title"],
        content=request.form["content"],
        author=current_user
    )
    session.add(entry)
    session.commit()
    return redirect(url_for("entries"))
```

Here you use the `current_user` variable from Flask-Login to set the author attribute when you create the entry. You can then add `{{ entry.author.name }}` to the metadata `div` in your `render_entry` macro in *templates/macros.html*.

Now try adding an entry then viewing it. You should see that your name is now displayed as author of the entry.

At this point it is simple to continue working with migrations. Each time you make changes to the database you should run `python manage.py db migrate` to generate a migration script. If the script looks correct then `python manage.py db upgrade` will apply the changes. If you need to roll back changes at any point then `python manage.py downgrade` will reverse a migration. And that's all there is to managing a changing database schema.

Why are migrations useful?

> *They allow us to apply changes to a database schema in a structured way. The migrations can be applied across environments, so you could run them on your development server, then run them on your production environment when they have been fully tested.*

What is the difference between Flask-Migrate's `migrate` and `upgrade` commands?

> *`migrate` only generates the migration script – it does not make any changes to the database. `upgrade` applies the most recent migration to the database.*

How would you change the actions which were due to take place in a migration?

> *You can manually add or remove actions from the migration script contained in migrations/versions.*

How would you roll back a database to an earlier migration?

> *The `downgrade` command will run through the downgrade functions to return the database to an earlier state.*

How do we access an object representing the currently logged in user?

> *The user can be accessed through the `current_user` variable.*

☆ ☆ ☆ ☆ ☆ · Report a typo or other issue

✔ Mark as completed

Previous Next