Define and Access the Database

The application will use a SQLite database to store users and posts. Python comes with built-in support for SQLite in the **sqlite3** module.

SQLite is convenient because it doesn't require setting up a separate database server and is built-in to Python. However, if concurrent requests try to write to the database at the same time, they will slow down as each write happens sequentially. Small applications won't notice this. Once you become big, you may want to switch to a different database.

The tutorial doesn't go into detail about SQL. If you are not familiar with it, the SQLite docs describe the language.

Connect to the Database

The first thing to do when working with a SQLite database (and most other Python database libraries) is to create a connection to it. Any queries and operations are performed using the connection, which is closed after the work is finished.

In web applications this connection is typically tied to the request. It is created at some point when handling a request, and closed before the response is sent.

flaskr/db.py

```
import sqlite3

import click
from flask import current_app, g
from flask.cli import with_appcontext


def get_db():
    if 'db' not in g:
        g.db = sqlite3.connect(
            current_app.config['DATABASE'],
            detect_types=sqlite3.PARSE_DECLTYPES
        )
        g.db.row_factory = sqlite3.Row

    return g.db


def close_db(e=None):
    db = g.pop('db', None)
```

```
    if db is not None:
        db.close()
```

`g` is a special object that is unique for each request. It is used to store data that might be accessed by multiple functions during the request. The connection is stored and reused instead of creating a new connection if `get_db` is called a second time in the same request.

`current_app` is another special object that points to the Flask application handling the request. Since you used an application factory, there is no application object when writing the rest of your code. `get_db` will be called when the application has been created and is handling a request, so `current_app` can be used.

`sqlite3.connect()` establishes a connection to the file pointed at by the `DATABASE` configuration key. This file doesn't have to exist yet, and won't until you initialize the database later.

`sqlite3.Row` tells the connection to return rows that behave like dicts. This allows accessing the columns by name.

`close_db` checks if a connection was created by checking if `g.db` was set. If the connection exists, it is closed. Further down you will tell your application about the `close_db` function in the application factory so that it is called after each request.

Create the Tables

In SQLite, data is stored in *tables* and *columns*. These need to be created before you can store and retrieve data. Flaskr will store users in the `user` table, and posts in the `post` table. Create a file with the SQL commands needed to create empty tables:

`flaskr/schema.sql`

```
DROP TABLE IF EXISTS user;
DROP TABLE IF EXISTS post;

CREATE TABLE user (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  username TEXT UNIQUE NOT NULL,
  password TEXT NOT NULL
);

CREATE TABLE post (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  author_id INTEGER NOT NULL,
  created TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
  title TEXT NOT NULL,
  body TEXT NOT NULL,
```

```
  FOREIGN KEY (author_id) REFERENCES user (id)
);
```

Add the Python functions that will run these SQL commands to the `db.py` file:

flaskr/db.py

```
def init_db():
    db = get_db()

    with current_app.open_resource('schema.sql') as f:
        db.executescript(f.read().decode('utf8'))


@click.command('init-db')
@with_appcontext
def init_db_command():
    """Clear the existing data and create new tables."""
    init_db()
    click.echo('Initialized the database.')
```

`open_resource()` opens a file relative to the `flaskr` package, which is useful since you won't necessarily know where that location is when deploying the application later. `get_db` returns a database connection, which is used to execute the commands read from the file.

`click.command()` defines a command line command called `init-db` that calls the `init_db` function and shows a success message to the user. You can read Command Line Interface to learn more about writing commands.

Register with the Application

The `close_db` and `init_db_command` functions need to be registered with the application instance, otherwise they won't be used by the application. However, since you're using a factory function, that instance isn't available when writing the functions. Instead, write a function that takes an application and does the registration.

flaskr/db.py

```
def init_app(app):
    app.teardown_appcontext(close_db)
    app.cli.add_command(init_db_command)
```

`app.teardown_appcontext()` tells Flask to call that function when cleaning up after returning the response.

`app.cli.add_command()` adds a new command that can be called with the `flask` command.

Import and call this function from the factory. Place the new code at the end of the factory function before returning the app.

`flaskr/__init__.py`

```
def create_app():
    app = ...
    # existing code omitted

    from . import db
    db.init_app(app)

    return app
```

Initialize the Database File

Now that `init-db` has been registered with the app, it can be called using the `flask` command, similar to the `run` command from the previous page.

Note:

If you're still running the server from the previous page, you can either stop the server, or run this command in a new terminal. If you use a new terminal, remember to change to your project directory and activate the env as described in Activate the environment. You'll also need to set `FLASK_APP` and `FLASK_ENV` as shown on the previous page.

Run the `init-db` command:

```
flask init-db
Initialized the database.
```

There will now be a `flaskr.sqlite` file in the `instance` folder in your project.

Continue to Blueprints and Views.