Database Layer Configuration

Contents

Overview

In this module, we will look at the different ways to configure a database layer with your Flask application.

In-built/Stand-alone

When configuring a database with your application you have 2 main options: in-built or remote.

An in-built database is simply a file that contains all the data you wish to save for your application.

It should only be used for developing and testing an application, as it has lower performance, security and resilience than using a dedicated database server.

Stand-alone databases run independently from your application, whether that be on your local machine or on the cloud.

In-built

The in-built database layer we will be using is called `sqlite`.

To use this in-built database layer with Flask you need to give a following database URI:

```
app.config['SQLALCHEMY_DATABASE_URI'] = "sqlite:///data.db"
```

With this configuration, SQLAlchemy will create a `data.db` file in the same directory as the Python module that references it. This file will contain any data you want to persist.

Stand-alone

When using a database that is separate to your application there's a few more things to consider.

Creation

When using a stand-alone database you will need to ensure an empty database exists before our Flask applications can connect to it.

Connector

To use a database that is not part of the app we need to establish a connection between them.

The first thing we need to do this is a connector, and this connector will differ depending on the type of database you are using.

For MySQL databases, the connector is called `pymysql`. You can install it with:

```
pip3 install pymysql
```

Connection String

Now we have a connector, we can use it to establish a connection between the application and the database using a URI. For a `mysql` database it would look like this.

```
app.config['SQLALCHEMY_DATABASE_URI'] =
"mysql+pymysql://username:password@host/database_name"
```

> Note: You may need to configure your database server's firewall rules to allow your application to access it.

Using Environment Variables

This URI contains sensitive information (username, password) that would allow anyone who sees it access to your database. As a result, we should avoid hard-coding this information into our applications before we commit it to our Git history. We can do so using an **environment variable**.

Environment variables are set on the command line during the terminal session (i.e. from the time you start the terminal to when you close it). You can set an environment variable in bash using:

```
export VARIABLE=value
```

We can then reference an environment variable within our Python code using the built-in `os` module:

```
# though os is built in, you still need to import it
import os

app.config['SQLALCHEMY_DATABASE_URI'] = os.getenv("DATABASE_URI")
```

The `getenv()` method can then be used to find the value of an environment variable as specified by the string you pass through as an argument. In the above instance, we are setting the `'SQLALCHEMY_DATABASE_URI'` to be the value of the environment variable `DATABASE_URI`.

Then, before running the Python app, you would set the `DATABASE_URI` environment variable on your terminal like so:

```
export DATABASE_URI=mysql+pymysql://username:password@host/database_name
```

This way, the database URI information is scoped only to the terminal session and will not appear when you push your code to your repository.

Creating the Database Object

To interact with our database using SQLAlchemy, we need to instantiate an SQLAlchemy object. This acts as our interface between the Flask application and the database itself.

We first need to install SQLAlchemy, which we can do with pip:

```
pip3 install flask_sqlalchemy
```

We can then import the SQLAlchemy class into our code to instantiate the database layer.

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] =
"mysql+pymysql://username:password@host/database_name"

db = SQLAlchemy(app)
```

Once our Flask `app` object has been instantiated and its URI configuration has been set, we can create a `db` object. This will hold all of our data for when we need it.

Notice we pass our app object into the SQLAlchemy class. This is so the `db` object knows that the database we are using is going to work with this app object.

With this `db` object we can create the database schema with the method:

```
db.create_all()
```

This will generate the table schema as defined within the Flask code on the database side. This means we don't have to manually build the schema using SQL syntax.

> Note: If you are using an in-built database layer, this command will create the data.db file for you.

You can also delete the database schema with:

```
db.drop_all()
```

Tutorial

In this tutorial, we are going to create an in-built database layer with a simple Flask application.

Setup

This tutorial assumes you are working on an Ubuntu VM, at least version `18.04 LTS`.

First, install apt dependencies:

```
sudo apt update
sudo apt install python3 python3-venv python3-pip
```

Create a directory named `flask-db-layer` and make it your current working directory:

```
mkdir flask-db-layer && cd $_
```

We now need to create a Python virtual environment to install our pip requirements in. Create a new virtual environment named `venv` and activate it:

```
python3 -m venv venv
source venv/bin/activate
```

We need to make sure Flask and SQLAlchemy are installed:

```
pip3 install flask flask_sqlalchemy
```

Create the app

Create two files named `app.py` and `create.py`:

```
touch app.py create.py
```

Paste the following code into `app.py`:

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)

app.config["SQLALCHEMY_DATABASE_URI"] = "sqlite:///data.db"

db = SQLAlchemy(app)

if __name__ == "__main__":
    app.run(debug=True)
```

This code simply creates a Flask application with a connection to an sqlite database. At the moment, no table schema has been defined and the app has no additional functionality.

Next, paste the following code into `create.py`:

```
from app import db

db.create_all()
```

This script can be used to create the database schema (and the database itself, if it's an sqlite database as in this example). Separating it into its own script allows us to generate the schema manually.

Now you can run `create.py` and it will create your `data.db` file.

```
python3 create.py
```

> Note: You may see a warning that looks similar to this:
>
> ```
> /home/harryvolker/flask/db-layer/venv/lib/python3.6/site-
> packages/flask_sqlalchemy/__init__.py:873: FSADeprecationWarning:
> SQLALCHEMY_TRACK_MODIFICATIONS adds significant overhead and will be disabled
> by default in the future.  Set it to True or False to suppress this warning.
> ```
>
> Don't be alarmed! The process has been successful and you can ignore the warning. If you want to suppress the warning, you can add this line to your code:
>
> ```
> app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
> ```

Now run the command.

```
ls
```

And you should see your new file `data.db`

Clean Up

To stop your Flask application running, navigate back to your terminal and press `Ctrl+C`. You should now have control over your terminal again.

To deactivate the virtual environment, run:

```
deactivate
```

If you wish to delete the virtual environment, run:

```
rm -rf venv
```

Exercises

Now using the connector `pymysql`, try to connect to a separate database instance. To confirm that the connection was successful, the command `db.create_all()` should return no errors (you can ignore any 'warnings').

You can also print the `db` object to the Python console to see information about the database instance it is connected to.

▶ Hint