Monday: User Model

Creating The User model.

Models are a way of abstracting and giving a common interface to data. We can now start creating a model to define our user data.

models.py

```
from . import db

#...

class User(db.Model):
    __tablename__ = 'users'
    id = db.Column(db.Integer,primary_key = True)
    username = db.Column(db.String(255))

    def __repr__(self):
        return f'User {self.username}'
```

We create a `User` class that will help us create new users. We pass in `db.Model` as an argument. This will connect our class to our database and allow communication.

`__tablename__` variable allows us to give the tables in our database proper names. If not used SQLAlchemy will assume that the tablename is the lowercase of the class name.

Since a single user will represent a row in our table. We create columns using the `db.Column` class which will represent a single column. We pass in the type of the data to be stored as the first argument. `db.Integer` specifies the data in that column should be an Integer. Every row must have a primary key set to it. By default the `Column` class has `primary_key` set to `False`. We want to set it to `True` on the id column to set it as the `primary_key` column.

The `db.String` class specifies the data in that column should be a string with a maximum of 255 characters.

The `__repr__` method is not really important. It makes it easier to debug our applications.

Flask-script shell

Flask script allows us to create a Python shell inside our application. It will be useful to test features in our app and for debugging. We create this shell inside the _manage.py_ script

manage.py

```
from app import create_app,db
# .....
from app.models import User
# ...
```

We first import the `db` instance from the `app/__init__` file. We then import the `User` class from the *app/models.py* file.

```
#....
@manager.shell
def make_shell_context():
    return dict(app = app,db = db,User = User )
if __name__ == '__main__':
    manager.run()
```

We then use the `@manage.shell` decorator to create a shell context and the `make_shell_context` function allows us to pass in some properties into our shell. We return the `app` application instance, `db` database instance and the `User` class.

We can access the shell by running this command on the terminal:

```
(virtual)$ python3.6 manage.py shell
>>> db
<SQLAlchemy engine=postgresql+psycopg2://james:***@localhost/watchlist>
```

We can confirm that our items have been passed into the shell by just typing them in and pressing enter. When we type `db` we get the database URI.

Creating and Destroying Tables

```
>>> db.create_all()
```

First we need to create the table in our database. We run the `db.create_all` command to create tables and columns in our database.

Unfortunately SQLAlchemy does not have a way to automatically update when we have changed the `schema` of our database. We have to drop everything and start again. To do this we run the command:

```
>>> db.drop_all()
```

CRUD using SQLAlchemy

Every database follows the CRUD principles. 1. Create 2. Read 3. Update 4. Delete

1) Create

We can now create users and insert them into our database. But first, we need to create the table in our database again since we dropped it.

```
>>> db.create_all()
>>> user_james = User(username = 'James')
>>> user_christine = User(username = 'Christine')
```

We define two user models now we are ready to save them to our database. Notice we do not pass in the `id` parameter. This is because this is a `primary_key` value that will auto increment as we keep adding users.

```
>>> db.session.add(user_james)
>>> db.session.add(user_christine)
>>>
```

SQLAlchemy uses sessions as a storage location for our database changes. They mark changes as ready for saving and committing.

We then need to commit the data and save the changes to our database

```
>>> db.session.commit()
```

We use the session commit method to save our changes to the database. We can confirm all the changes have been saved when we go to the PostgreSQL server

```
#watchlist
select * from users;
 id | username
----+----------
1   |  James
2   |  Christine
```

Adding users one at a time to our sessions can be tedious. Luckily there is an easier way to do it.

```
>>> user_lisa = User(username = 'lisa')
>>> user_victor = User(username = 'victor')
>>> db.session.add_all([user_lisa,user_victor])
>>> db.session.commit()
```

2) Read

We can now start querying our database to retrieve saved data.

a) all() query.

We use the `all()` query function to get all the items in the model.

```
>>> users = User.query.all()
>>> users
[User James, User Christine, User lisa, User victor]
>>>
```

The `.all()` function returns a list of all the entries listed with that model.

b) first query.

We can also get the a single entry from our model.

```
>>> single_user = User.query.first()
>>> single_user
User James
>>>
```

We use the `first()` query function to get the first entry of our model.

c) filter_by query

We can also filter out data we are receiving using the `filter_by` query function.

```
>>> user = User.query.filter_by(username = 'James').first()
>>> user
User James
```

We pass in what we want to filter as an argument then we can choose whether to get all the items or just the first item.

d) filter query

This is used as an alternative to the `filter_by` query.

```
>>> users = User.query.filter(User.id > 1).all()
>>> users
[User Christine, User lisa, User victor]
```

Notice how the `User James` instance is not displayed. This is because we started filtering our data from ids' that are greater than one.

e) limit query

We can also limit the amount of responses we get.

```
 >>> users = User.query.limit(2).all()
>>> users
[User James, User Christine]
```

We use the limit query and pass in how many queries we need.

3) Update

We can also use SQLAlchemy to update entries from our database.

```
>>> single_user = User.query.filter_by(id = 1).update({"username": "James Muriuki"})
>>> db.session.commit()
```

Here we filter a query by getting the entry with the `id` of 1 and we call in the update function. We then pass in the property we want to update with the new value.

Finally we have to commit any changes we have in our database. We can confirm that the entry has been changed from the PostgreSQL server.

```
watchlist=# select * from users;
 id |   username
----+---------------
  2 | Christine
  3 | lisa
  4 | victor
  1 | James Muriuki
(4 rows)
```

4) Delete

We can also remove entries from the database.

```
>>> single_user = User.query.filter_by(id = 1).first()
>>> single_user
User James Muriuki
>>> db.session.delete(single_user)
>>> db.session.commit()
>>>
```

We use the session delete function that we pass in the entry to delete. We finally commit the changes to the database. We can confirm that the entry has been removed from the PostgreSQL server.

```
watchlist=# select * from users;
 id | username
----+-----------
  2 | Christine
  3 | lisa
  4 | victor
(3 rows)
```

You can find our application at this point from here https://github.com/mbuthiya/watchlist/tree/18-creating-first-model