Monday: Migration; Password Hashing

Database Migrations

Functionality of web apps changes and we need to track changes to our database. In our small application it is easy to just drop our database when we change the schema of our database. But when we have a large application we can't really afford to lose data everytime we change the structure of our models.

Alembic

Alembic is a tool that automatically creates and tracks **database migrations** from changes to our SQL models. Database migrations are records of all the changes of our database schema.

Alembic allows us to upgrade or downgrade our database to saved versions

Flask-Migrate

We will use the flask-migrate extension to create database migrations

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

We use pip to install the `flask-migrate` extension.

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

Next we will drop our current database schema and use Flask-Migrate to create a new schema. We then need to initialize the flask extension

manage.py

```
#...
from  flask_migrate import Migrate, MigrateCommand
#...
```

We first import `Migrate` class and the `MigrateCommand` class.

```
#.......

migrate = Migrate(app,db)
manager.add_command('db',MigrateCommand)
#......
```

We then initialize the `Migrate` class and pass in the `app` instance and the `db` SQLAlchemy instance. We then create a new manager command `'db'` and pass in the `MigrateCommand` class.

Initializing

To use migrations we first need to Initialize our application to use Migrations.

```
(virtual)$ python3.6 manage.py db init
```

This will create a migrations folder in our root directory. This will be where our migration versions will be stored.

We can now create the first migration

```
(virtual)$ python3.6 manage.py db migrate -m "Initial Migration"
```

This will create a migration version.We can now upgrade to that version of the database.

```
(virtual)$ python3.6 manage.py db upgrade
```

We can go to our psql server and see that all the tables and columns of our database have been created.

You can find the application at this point here https://github.com/mbuthiya/watchlist/tree/20-Migrating-database

User Authentication

We now want to make sure our users are authenticated before they create a review on our application.

We first need to create a Column in our `users` table that stores our passwords .

models.py

```
#...
class User(db.Model):
    #...
    pass_secure = db.Column(db.String(255))
```

We then update our database by running our migrate commands to record the change in the database schema and to upgrade to the new schema with the added column.

Securing our passwords

The key to making our users' information secure in our application, is not to store their passwords but **hashes** of the passwords. Password hashing is the act of taking a password and performing a series of cryptographic transformations to it.

Hashed passwords can be verified in place of actual passwords because identical inputs produce an identical hash.

models.py

```
from werkzeug.security import generate_password_hash,check_password_hash
```

Flask's Werkzeug’s security module provides hashing functionality with 2 methods.

- **generating_password_hash** - This function takes in a password and generates a password hash.
- **check_password_hash** - This function takes in a hash password and a password entered by a user and checks if the password matches to return a True or False response.

We import these 2 functions and we will use them in our `User` model.

```
    pass_secure  = db.Column(db.String(255))

        @property
        def password(self):
            raise AttributeError('You cannot read the password attribute')

        @password.setter
        def password(self, password):
            self.pass_secure = generate_password_hash(password)


        def verify_password(self,password):
            return check_password_hash(self.pass_secure,password)
```

We use the `@property` decorator to create a write only class property `password`. When we set this property we generate a password hash and pass the hashed password as a value to the `pass_secure` column property to save to the database.

We raise an `AttributeError` to block access to the `password` property. This is because it is not secure for Users to have access to that property.

We create a method `verify_password` that takes in a password, hashes it and compares it to the hashed password to check if they are the same.

We can create a test file to test this. Create a new file `test_user.py` inside our tests folder.

test_user.py

```
import unittest
from app.models import User

class UserModelTest(unittest.TestCase):

    def setUp(self):
        self.new_user = User(password = 'banana')

    def test_password_setter(self):
        self.assertTrue(self.new_user.pass_secure is not None)
```

We create a `setUp` method that creates an instance of our `User` class we then pass in the `password` property. We then create a testcase `test_password_setter` this ascertains that when password is being hashed and the `pass_secure` contains a value.

test_user.py

```
    def test_no_access_password(self):
            with self.assertRaises(AttributeError):
                self.new_user.password

        def test_password_verification(self):
            self.assertTrue(self.new_user.verify_password('banana'))
```

The second taste case confirms that our application raises an `AttributeError` when we try and access the `password` property The third test confirms that our password_hash can be verified when we pass in the correct password.

Authentication Blueprint

We now need to create a blueprint that will handle all our application authentication requests. Create a Python Package `auth` inside our `app` package. Create an `__init__.py`. We also create a new _views.py__ file inside the `auth`package.

app/auth/__init__.py

```
from flask import Blueprint

auth = Blueprint('auth',__name__)

from . import views
```

Here we simply create a Blueprint instance `auth` and we import the `auth` views module.

app/auth/views.py

```
from flask import render_template
from . import auth

@auth.route('/login')
def login():
    return render_template('auth/login.html')
```

We then create a `login` view function that renders a template file. This template is stored in a subfolder inside the templates folder `templates/auth/login.html`

app/__init__.py

```
def create_app(config_name):
    #...
    from .auth import auth as auth_blueprint
    app.register_blueprint(auth_blueprint,url_prefix = '/authenticate')

    #....
```

We register our Blueprint instance in our `create_app` function. We pass in a `url_prefix` argument that will add a prefix to all the routes registered with that blueprint. We can access the login route `localhost:5000/authenticate/login`