This tutorial is a part of the **"Flask Users, Sessions and Authentication"** course on **Pluralsight**, by **Mateo Prigl**. Source code for each lesson is provided, don't forget to create your own database.

Introduction

Before taking the course you should be familiar with the following packages:
flask-sqlalchemy
flask-migrate
flask-wtf

Flask-sqlalchemy package is a wrapper around the SQLAlchemy, which is an object relational mapper for Python.
We will use it instead of relying on raw SQL queries.
Flask-migrate package is a wrapper around the Alembic package, which helps us in creating database migrations.

The course that you are taking will be a part of the Flask path.
One of the courses from this path will be on creating database models in flask using SQLAlchemy and Alembic, however, that course does not exist yet (at least not at the time of me writing this).
You could research these topics on your own, or you can read this tutorial pdf to get familiar with them.
This tutorial will also explain the structure of the application that we are going to build throughout the course. This means that you have to be aware of what the **application factory** is and what **blueprints** do.

To process web forms we will use the flask-wtf package, which is a wrapper around the WTForms toolkit.

> One more thing to note here! If you have time, check out my other course on Pluralsight, ***Creating and Processing Web Forms with Flask*** (link). In this course I talk about flask-wtf and WTForms, but what is also important is that some workflows I use in this course are taken "or similar" to that one. For example, I use two macros to print out form errors in every form. I use Bootstrap alerts to show flash messages from the base template. All of these things are also explained in that course. That being said, the courses before this one are not a requirement, just a suggestion.
> Ok, let's get to it.

Flask-sqlalchemy (SQLAlchemy)

SQLAlchemy is a Python SQL toolkit and ORM (Object Relational Mapper). Flask-sqlalchemy just provides us with some minor integration of this package to the framework.
Most of the applications will need some kind of a database to store information. If that database is relational, we would usually rely on SQL queries to manage it.
For example, maybe I have some MySQL database with the users table, which contains information about all of the users from the site. To get all of the information about the user with the name "mateo", I could write a query like this:

```
SELECT * FROM users WHERE username="mateo"
```

This query would select all of the columns from the users table, but only for the row that has a username column equal to "mateo". In Flask, like in other frameworks, we would usually have an engine session, some kind of an interface which would allow us to write these queries and execute them on the database. And of course, get the result and do something with it.
However, instead of writing these raw SQL queries, we could use an ORM. ORM will *map* the *relational* data to some *object*. This is more intuitive to use, since Python, like many other languages, is object oriented.

So instead of writing the query above, we could do something like:

```
User.query.filter_by(username="mateo")
```

So the `User` class represents the *users* table from the database. This class has methods which we can use to retrieve or edit information from the SQL tables. ORM provides a level of abstraction. We don't really have to know how it's done, just how the interface works. In the background, these methods will generate SQL queries by themselves.
Now let's see some basic things we can do with SQLAlchemy.

To use it in your project, you first have to install it:

```
pip install flask-sqlalchemy
```

If you are using it only for this course, it will be enough to just do:

```
pip install -r requirements.txt
```

This will install all of the needed packages, including the sqlalchemy.
To make it work, we first need to configure at least two configuration options:

```
SQLALCHEMY_DATABASE_URI='sqlite:///'+ os.path.join(basedir, 'globomantics.sqlite')
SQLALCHEMY_TRACK_MODIFICATIONS=False
```

These are the configuration options I took from the application we use in the course. I defined them inside of the app.config dictionary.
The first one defines an URL to the database, which in this case uses SQLite engine. Other option will make the transactions faster, since we don't need to track modifications.
Usually you would need to define a port, username, password or some other requirements to access some more complex database engines like MySQL or PostgreSQL, but not here, not for SQLite. This database will just be a simple file inside of the project structure.

Remember to always have the database created when trying out the application. I won't provide a created database in every source file. This is something you have to do manually, but we will get to that.

To have an access to the database, you would instantiate the `db` object inside of the source code:

```
from flask_sqlalchemy import SLQAlchemy


db = SQLAlchemy(app)
```

Notice that I passed the `app` Flask instance to the class. This will connect the database session to the current application.
This is how you would do it if you don't use an application factory.

And now we can define some database tables, by defining some ORM classes.

```
class User(db.Model):
    __tablename__ = 'users'

    id = db.Column(db.Integer(), primary_key=True)
    username = db.Column(db.String(64), unique=True, index=True)

    def __init__(self, username):
        self.username = username

    def __repr__(self):
        return '<User %r>' % self.username
```

What we defined here is a simple *users* table.
The `User` class inherits the functionality from the `db.Model` class, provided by SQLAlchemy. The `__tablename__` attribute defines the name of the table which will be mapped to this object from the

database. This table will have two columns, which are defined like properties of the `User` class. Also each column is an instance of `db.Column`. `db.Integer()` and `db.String(64)` define the type of the column, and we have some other options like primary key and indexes, which should be familiar from SQL.
`__repr__` method has nothing to do with SQLAlchemy, it just says how the object will be printed out if we try to do it.
The `__init__` method defines how we need to initialize new instances. We only need to provide the username.

Now we can use some type of an interactive python shell, maybe even the Flask shell itself.

```
$ flask shell

>> from app import db
>> db.create_all()
>>
```

I entered the flask shell and imported the `db` instance from wherever we created it. Then I used the `create_all` method to create the database from the given schema, defined with the `User` class. This will create the actual database and the *users* table. We will do this differently, because we use database migrations, but this is also a quick way to create it without the use of migrations.
If you are trying out the database and learning how the SQLAlchemy works, that means that you will probably use the shell a lot. But that means that you need to import `db` and `User` every time you open the shell. And if you have more models, you have to import all of them. Instead of doint that, you could define a shell context processor inside of the main application file:

```
@app.shell_context_processor
def make_shell_context():
    return dict(db=db, User=User)
```

This is what I did for the course application. The `shell_context_processor` decorator will make the returned dictionary available to the Flask shell. These two parameters will then be available to you every time you use it. Isn't that great? I think it's useful.

Now let's see the usage of the package, finally.
To create a new user using the `User` class, we can do it like this (from the shell or from the source code):

```
user = User('mateo')
db.session.add(user)
db.session.commit()
```

I created a new user with the `User` model class. I only provided the username, because that is how we defined the `__init__` method and also the username is the only column we really need to provide (id is self generated and autoincremented). When you instantiate a new object, that doesn't mean that it exists in the database. You need to first add it to the database session and then commit the changes. Only then the new row in the database table is created. Until then you can treat this instance as any other instance in Python.

Now let's say that I want to modify that same user and give it another name.

```
user.username = "other name"
db.session.add(user)
db.session.commit()
```

This is logical, nothing revolutionary here. To delete the user, we can do.

```
db.session.delete(user)
db.session.commit()
```

Of course, to do all of these things you would need to have the same user instance you created. What about fetching users from the database? To do this you can use the `query` object that is attached to every model.

```
user = User.query.filter_by(username="mateo").first()
```

This will get the "mateo" user from the database and put it inside of the user variable. Notice the `first` method at the end. This is one of the *query executors*. These will say what you want to returned after the query is executed. The `first` means that you only want to get one user instance, the first one of the matched, which in this case will be one, since the username column needs to have unique usernames. `all` will return a list of all users that match the certain query. You can also use the `get` method, by only when you know the id of the user.

```
user = User.query.get(1)
```

This will get the user with the id=1 and in this case you don't need the query executor.
Having only one model (table) is boring and not really useful. Usually we will have more tables which are interconnected, which means that they are related to each other in some way. We need to define these relationships in the model classes.
For example, one to many relationship between the two models means that one model will have a lot of rows which are in same way related to only one row in other model (hence the name, one to many).

For example, one role could have many users. When I say role, I mean like, if there is an Admin role in the database, a lot of users could have that role, because a lot of users could be administrators.
This relationship could be defined like this:

```
class Role(db.Model):

    users = db.relationship('User', backref='role', lazy='dynamic')

class User(db.Model):

    role_id = db.Column(db.Integer(), db.ForeignKey('roles.id'))
```

Of course, these models would have more columns, but I didn't write them in here. We are only interested in the relationship between these to. So the user can only have one role, which means that it has a `role_id` field. This field contains the id of the role from the *roles* database. Role doesn't need any additional columns, because it is placed at the *many* side of the *one to many* relationship. *role_id* is a foreign key to the foreign *roles* database and it refers to the *id* column of that database. That is how we connect these to models (or tables).
Inside of the *Role* model we can define the relationship. Role instance will have the *users* property, which will get back all of the users with the *role_id* of the current role instance we execute it on. So we can do:

```
role = Role.query.filter_by(name="admin").first()
role.users.all()
```

This will get the admin role instance from the database. Then when we do the `role.users`, this would get back all of the users that have that admin role.
The `lazy='dynamic'` means that we always want to define the query executor, which is why I had to add `all()`. You may see this as an inconvenience, but it gives us more power over how the relationship is handled.
The `backref='role'` defines the relationship from the users perspective. It means that we can also do:

```
user = User.query.get(1)
user.role
```

This will get us back the role instance of the role that the user has. If we did not do this, we would have to do something like this:

```
user = User.query.get(1)
role_id = user.role_id
Role.query.get(role_id)
```

If you want to define a *many to many* relationship, you would need an additional *join* table. Depending on the complexity of the relationship this table will be a model, or just a simple table with the ids of the both models from the relationship. We will do this in the course, by defining the relationship between musicians

and gigs through the applications table. Since we are going to cover this later in the course, I will not write about it here.
To add users to some role, we can do:

```
role.users.append(user)
db.session.add(role)

# Or to remove a user
role.users.remove(user)
```

I also want to mention one option that we will use in the course. The *cascade* option.

```
remember_hashes    = db.relationship("Remember", backref="user", lazy="dynamic", \
cascade="all, delete-orphan")
```

This is one of the lines that we will define at the end of the first module.
cascade="all, delete-orphan" means, keep all of the default cascading options, but also add *delete-orphan* which will delete all of the related instances in the connected database. Basically, it means, delete all of the related instances defined by a relationship, when an instance is deleted from the database.

For more information check out the SQLAlchemy site (link) or the Flask-SQLAlchemy site (link).

Flask-Migrate (Alembic)

We need a systematic way for updating the database. Imagine that your application is in the production and you want to update a column in some of the tables from the database. Or maybe you are developing your application with more developers and all of them need to have the same version of the database. Git helps us in version control of the code. Migrations are like git commits for databases. Pythons Alembic is one of the more popular options for managing database migrations. Flask-migrate is a wrapper for Alembic in Flask.

Of course, install it first with pip (this is also included in the requirements.txt file).
Then we can initialize it:

```
from flask_migrate import Migrate

migrate = Migrate(app, db)
```

We need to pass the database connection and also the current Flask application.
By installing the package we have the access to the `flask db` set of commands.

First thing you need to do in a new project is to initialize it (this will be done in the "Application Structure" lesson).

```
flask db init
```

This will create the **migrations** folder inside of the root directory of the project.
We don't need to know how this is implemented, you are only interested in the **versions** subfolder. This folder contains all of the database versions, all of the migration scripts that do the actual change on the database. You can write them manually, but there is a better way.

```
flask db migrate
```

This command will take a look at all of the models you defined in the project. It will analyze them and their columns and based on that it will automatically generate the migration script inside of the **versions** folder. After that you only need to do:

```
flask db upgrade
```

This will execute all of the migration scripts. We should create a new script for every change we need to do inside of the database.
We will follow that practice in this course.
If you made a mistake or you want to start fresh, delete the database and start the migration again.
Later in the course, when we create the application, I will create a python script which will randomly populate the database, so that we don't have to create users and other instance manually when we want to test the website.

To learn more, check out the Flask-Migrate package (link).

Flask-WTF (WTForms)

To process forms, we will us the Flask-WTF package. Every form is represented like a class and it needs to extend the `FlaskForm` class. The package needs to be installed with *pip*, but of course, it is also included in the starting *requirements.txt* file. Let's take a look at the registration form example, which is already included in the application.

```
from flask_wtf import FlaskForm
from wtforms.fields import StringField, PasswordField, SubmitField, \
BooleanField, RadioField, TextAreaField
from wtforms.fields.html5 import EmailField
from wtforms.validators import InputRequired, DataRequired, EqualTo, Length, ValidationError, Email
from app.models import User

class RegistrationForm(FlaskForm):
    username         = StringField("Username *",
                               validators=[
                                   InputRequired("Input is required!"),
                                   DataRequired("Data is required!"),
                                   Length(min=5, max=20, message="Username \
                          must be between 5 and 20 characters long")
                               ])
    email            = EmailField("Email *",
                               validators=[
                                   InputRequired("Input is required!"),
                                   DataRequired("Data is required!"),
                                   Length(min=10, max=30, message="Email must be \
                               between 5 and 30 characters long"),
                                   Email("You did not enter a valid email!")
                               ])
    password         = PasswordField("Password *",
                               validators=[
                                   InputRequired("Input is required!"),
                                   DataRequired("Data is required!"),
                                   Length(min=10, max=40, message="Password must \
                                be between 10 and 40 characters long"),
                                   EqualTo("password_confirm", message="Passwords must match")
                               ])
    password_confirm = PasswordField("Confirm Password *",
                               validators=[
                                   InputRequired("Input is required!"),
                                   DataRequired("Data is required!")
                               ])
```

```
    location         = StringField("Your location (e.g. city, country)",
                               validators=[
                                   InputRequired("Input is required!"),
                                   DataRequired("Data is required!"),
                                   Length(min=3, max=40, message="Location\
                              must be between 3 and 40 characters long")
                               ])
    description      = TextAreaField("Description *",
                               validators=[
                                   InputRequired("Input is required!"),
                                   DataRequired("Data is required!"),
                                   Length(min=10, max=200, message="Description\
                              must be between 10 and 200 characters long")
                               ])
    submit           = SubmitField("Register")

    def validate_username(form, field):
        user = User.query.filter_by(username=field.data).first()
        if user:
            raise ValidationError("Username already exists.")

    def validate_email(form, field):
        user = User.query.filter_by(email=field.data).first()
        if user:
            raise ValidationError("Email already exists.")
```

`RegistrationForm` is extending the `FlaskForm`. Each property of this form class is like a web form field. Each field is also created by the classes like `StringField` which we take directly from the *flask.wtforms* package.
The first argument will be the label of the field. The second one is the keyword argument for all of the validators, the things that need to be valid for this form to be processed successfully.
I also defined two extra validators down in the class, which are methods that accept the form and the field that they will operate on. Form knows which field I'm talking about because of the field name after the *validate_*.
This is one of the ways to define validators.
Email validator will just check the basic structure of the string input, if it looks like an email. Even though this is the included wtforms validator, you still need to install an additional *email-validator* package. This is included in the *requirements.txt* file, but be aware of this when you are creating your own applications.

After doing this you need to initialize the form in the view and pass it to the template.

```
@auth.route("/register", methods=["GET", "POST"])
def register():
    form = RegistrationForm()

    if form.validate_on_submit():
        username    = form.username.data
        email       = form.email.data
        password    = form.password.data
        location    = form.location.data
        description = form.description.data

        user = User(username, email, password, location, description)
        db.session.add(user)
        db.session.commit()
```

```
        flash("You are registered", "success")
        return redirect(url_for("main.home"))

    return render_template("register.html", form=form)
```

This is the usual workflow when working with forms. You first initialize it in the *form* object. Than you pass it to the html template which will show the form. Of course this view can also be accessed with *POST* method, which means that someone tried to submit the form. In that case, the *validate_on_submit* method will check if all of the input is valid. Only if all of the fields are validated and the form is *POST*ed, the condition will be true. Then we can take the input with the *form.field_name.data*. Here I use the *User* SQLAlchemy model to create a new user and redirect the new user to the home page.
Of course, this form needs to be rendered in the *register.html* template.

```
{% extends 'base.html' %}
{% block title %}Register {% endblock %}
{% from '_error_messages.html' import field_error_messages, error_messages %}
{% set active_page = 'register' %}

{% block content %}
<div class="row">
  <div class="col-lg-7 offset-lg-2 my-5">
    <h1>Register</h1><hr>
    <form method="POST" action="{{ url_for('auth.register') }}">
      {{ form.csrf_token }}
      {{ error_messages(form.errors) }}
      <div class="form-group">
        {{ form.username.label }}
        {{ form.username(class="form-control") }}
        {{ field_error_messages(form.username) }}
      </div>
      <div class="form-group">
        {{ form.email.label }}
        {{ form.email(class="form-control") }}
        {{ field_error_messages(form.email) }}
      </div>
      <div class="form-group">
        {{ form.password.label }}
        {{ form.password(class="form-control") }}
        {{ field_error_messages(form.password) }}
      </div>
      <div class="form-group">
        {{ form.password_confirm.label }}
        {{ form.password_confirm(class="form-control") }}
        {{ field_error_messages(form.password_confirm) }}
      </div>
      <div class="form-group">
        {{ form.location.label }}
        {{ form.location(class="form-control") }}
        {{ field_error_messages(form.location) }}
      </div>
      <div class="form-group">
        {{ form.description.label }}
        {{ form.description(class="form-control", rows="8") }}
        {{ field_error_messages(form.description) }}
      </div>
```

```
      <hr>
      <div class="form-group">
        {{ form.submit(class="btn btn-primary form-control") }}
      </div>
    </form>
  </div>
</div>
{% endblock %}
```

You can render each field just by calling on it. Also, you can render the label by appending *label* to the field instance. I imported two macros from other file to print out the errors. If the form did not pass the validation inside of the view, it will contain all of the validation errors inside of the *form.errors* dictionary. I use these two macros *field_error_messages* and *error_messages* to access these errors and show them if they exist.

If you are interested in how this works, how custom validators work, or how to create these error macros, check out my other course on Pluralsight, on forms (link).

To see how I created these macros look for the **Demo: Show Errors with Jinja Macros** lesson in the last module of that course.

Application Structure

Flask is a micro-framework. It provides us with just enough functionality to start creating web application, but all of the other functionality needs to be extended. This is also true for the application structure. It is on you to decide how you want to organize the source code. One of the recommended ways for larger applications is to use the **application factory** and **blueprints**.

We are starting the application from the **setup.py** file.

```
from app import create_app, db
from app.models import User
from flask_migrate import Migrate

app = create_app()
migrate = Migrate(app, db)

@app.shell_context_processor
def make_shell_context():
    return dict(db=db, User=User)
```

This file will instantiate the application with the **create_app** application factory. Our factory will be simple, you can extend it in more ways if your application gets more complicated. For that you can check the Flask website, or you can watch a course on Pluralsight. I think there will be one on creating the Flask application structure.

So this file will also initialize the migrations from the flask-migrate package. Now let's take a look at the application factory which actually defines the application. It is placed inside of the **app** package **init.py** file.

```
import os
from flask import Flask, render_template
from flask_sqlalchemy import SQLAlchemy

basedir = os.path.abspath(os.path.dirname(__file__))
db = SQLAlchemy()
```

```
def create_app():
    app = Flask(__name__)
    app.config.from_mapping(
        SECRET_KEY=os.getenv("FLASK_SECRET_KEY") or 'prc9FWjeLYh_KsPGm0vJcg',
        SQLALCHEMY_DATABASE_URI='sqlite:///'+ os.path.join(basedir, 'globomantics.sqlite'),
        SQLALCHEMY_TRACK_MODIFICATIONS=False,
        DEBUG=True
    )

    db.init_app(app)

    from app.auth.views import auth
    from app.main.views import main
    app.register_blueprint(auth)
    app.register_blueprint(main)

    from app.main.errors import page_not_found
    app.register_error_handler(404, page_not_found)

    return app
```

Here we define the *create_app*. It first creates the application instance from the **Flask** class, which should be familiar. Then we define the configuration options for SQLAlchemy and the secret key (we need this key for encryption, csrf tokens, flask-wtf, sessions...).

Notice that I declared the *db* instance outside of the application factory. This is because I want to import these instance in all of the other files, so the *db* variable needs to be global in this file. However, that instance gets initialized inside of the application factory `db.init_app(app)`. This will connect the database to the application.

Then we register blueprints we created in the *auth* and *main* packages *views.py* file.
Let's take look at the *main/views.py* file.

```
from flask import Blueprint, render_template
from app.models import User

main = Blueprint('main', __name__, template_folder='templates')

@main.route('/')
def home():
    users = User.query.all()
    return render_template('home.html', users=users)
```

Here I define the *main* blueprint we registered inside of the application factory. The blueprint is create with the *Blueprint* class. Blueprints will let you divide your application in smaller logical pieces. Instead of defining every route as **@app.route**, you can separate the logic of the *main* sites with the *main* blueprint and to **@main.route** instead.
template_folder argument will just say Flask to also look inside of the *main/templates* package folder for templates and not just inside of the main *templates* folder in the *app* package.
When you register these blueprints, you can also add the *url_prefix* option which will prepend something to the url's of all the views inside of the blueprint. For example, if I define a route in the *main* blueprint as `@main.route("/hello")` and also add the *url_prefix="/hi"* to the blueprint registration, the resulting url for that view will be **/hi/hello**. This **/hi** will be prepended to all of the blueprint routes.

Models are defined inside of the **models.py** module of the *app* package. There are also *templates* and *static* folder which contains all of the shared static files and templates.
Each part of the application will have its separate package and it will define the blueprint inside of the

views.py file. If it has forms, it will define them in the *forms.py* file and import them from there.

And that's it, that is what you need to know for now. All of the rest will be explained throughout the course. Hope that this tutorial helped you a little, or at least guided you to the right path.

Thanks for watching the course!
Cheers
Mateo