

Discussion 8

Author: Zhe Wang

01D Sit with your project group in sections by your assigned TA

Front of Classroom

Joon Young	Samia
Josh/Alex	Alexandra
Zach	Harris
Joyce	Danny

02D Sit with your project group in sections by your assigned TA

Front of Classroom

Jason	Samy
Josh/Alex	Neel
Han	Justin
Samia	Konstantinos

MVC Architecture

- Model

The central component of the pattern. It is the application's dynamic data structure, independent of the user interface. It directly manages the data, logic and rules of the application.

- View

Any representation of information such as a chart, diagram or table or web UI.

- Controller

Accepts input and converts it to commands for the model or view.

Controller

- Backend Endpoint

- Handling parsing HTTP requests send from frontend
- Translate requests to SQL query
- Send SQL query via connector
- Translating database records to Data Access Objects (DAOs)(Python Objects)
- Returning HTTP response

```
@bp.route('/login', methods=['GET', 'POST'])
def login():
    if current_user.is_authenticated:
        return redirect(url_for('index.index'))
    form = LoginForm()
    if form.validate_on_submit():
        user = User.get_by_auth(form.email.data, form.password.data)
        if user is None:
            flash('Invalid email or password')
            return redirect(url_for('users.login'))
        login_user(user)
        next_page = request.args.get('next')
        if not next_page or url_parse(next_page).netloc != '':
            next_page = url_for('index.index')

        return redirect(next_page)
    return render_template('login.html', title='Sign In', form=form)
```

View

- Web UI
- Could be separate frontend like React, Vue.js, Angular or could be server side rendering returning HTML to browser to render such as Django
- Flask provides server side rendering with `render_template` that takes in [Jinja template](#) with context variables then returns a rendered HTML wrapped in HTTP Response

```
<form action="" method="post" novalidate>
  {{ form.hidden_tag() }}
  <p>
    {{ form.email.label }}<br/>
    {{ form.email(size=32) }}<br/>
    {% for error in form.email.errors %}
      <span style="color: red;">{{ error }}</span>
    {% endfor %}
  </p>
  <p>
    {{ form.password.label }}<br/>
    {{ form.password(size=32) }}<br/>
    {% for error in form.password.errors %}
      <span style="color: red;">{{ error }}</span>
    {% endfor %}
  </p>
  <p>
    {% with messages = get_flashed_messages() %}
    {% if messages %}
      <ul>
        {% for message in messages %}
          <li>{{ message }}</li>
        {% endfor %}
      </ul>
    {% endif %}
    {% endwith %}
  </p>
  {{ form.submit(class="btn btn-black") }}
  <a class="btn btn-secondary" href="{{ url_for('users.register') }}" role="button">Register</a>
</form>

{% endblock %}
```

Model

- Usually represented by classes or objects depending on language
- Containing different methods for different business needs
- Translate between database table rows
- provides an abstract interface to some type of database or other persistence mechanism

```
class User(UserMixin):
    def __init__(self, id, email, firstname, lastname):
        self.id = id
        self.email = email
        self.firstname = firstname
        self.lastname = lastname

    @staticmethod
    def get_by_auth(email, password):
        rows = app.db.execute("""
SELECT password, id, email, firstname, lastname
FROM Users
WHERE email = :email
""",
                               email=email)
        if not rows: # email not found
            return None
        elif not check_password_hash(rows[0][0], password):
            # incorrect password
            return None
        else:
            return User(*(rows[0][1:]))
```

SQLAlchemy

- SQL Connector and Object Relational Mapper (ORM)
- Open up connection to PostgreSQL
 - Execute SQL Query
 - Return Database Records
 - Map Database Records to Python Classes/Objects

```
from sqlalchemy import create_engine, text

class DB:
    """Hosts all functions for querying the database.

    Use the execute() method if you want to execute a single SQL
    statement (which will be in a transaction by itself.

    If you want to execute multiple SQL statements in the same
    transaction, use the following pattern:

    >>> with app.db.engine.begin() as conn:
    >>>     # everything in this block executes as one transaction
    >>>     value = conn.execute(text('SELECT...'), bar='foo').first()[0]
    >>>     conn.execute(text('INSERT...'), par=value)
    >>>     conn.execute(text('UPDATE...'), par=value)
    >>>

    """

    def __init__(self, app):
        self.engine = create_engine(app.config['SQLALCHEMY_DATABASE_URI'],
                                     execution_options={"isolation_level": "SERIALIZABLE"})

    def execute(self, sqlstr, **kwargs):
        """Execute a single SQL statement sqlstr.
        If the statement is a query or a modification with a RETURNING clause,
        return the list of result tuples;
        if the statement is an UPDATE or DELETE statement without RETURNING
        """
```

Demo: A walk thru of an example project

- How to create table and load data
- Where is Model
- Where is Controller
 - What is FlaskForm ([flask-wtf](#))
 - What is API Endpoint, [URL Route Registration](#)
 - What is [render_template](#)
- Where is view
 - [Jinja template](#)

How does Flask Form Work?

- First, create a form class extending FlaskForm base class provided by flask_wtf
 - Add necessary form fields using classes from wtforms under the class
- Then in HTML template, insert a form tag block with jinja code block
 - Add all necessary fields and submit button if necessary
- Finally in the backend endpoint controller function, instantiate a new instance of the form then validate input and perform other logics with data parsed from the form

Creating Forms

Flask-WTF provides your Flask application integration with WTForms. For example:

```
from flask_wtf import FlaskForm
from wtforms import StringField
from wtforms.validators import DataRequired

class MyForm(FlaskForm):
    name = StringField('name', validators=[DataRequired()])
```

Note:

From version 0.9.0, Flask-WTF will not import anything from wtforms, you need to import fields from wtforms.

In addition, a CSRF token hidden field is created automatically. You can render this in your template:

```
<form method="POST" action="/">
    {{ form.csrf_token }}
    {{ form.name.label }} {{ form.name(size=20) }}
    <input type="submit" value="Go">
</form>
```

If your form has multiple hidden fields, you can render them in one block using `hidden_tag()`.

```
<form method="POST" action="/">
    {{ form.hidden_tag() }}
    {{ form.name.label }} {{ form.name(size=20) }}
    <input type="submit" value="Go">
</form>
```

Validating Forms

Validating the request in your view handlers:

```
@app.route('/submit', methods=['GET', 'POST'])
def submit():
    form = MyForm()
    if form.validate_on_submit():
        return redirect('/success')
    return render_template('submit.html', form=form)
```

How does HTTP work?

- The HTTP route for a service is the publicly-accessible directory path that maps to the root of your service. (xxxx.com/abc/1/0)
- HTTP Request has different type: GET, POST, PUT, PATCH, and DELETE
- The browser sends HTTP requests to the flask backend
- The flask backend sends HTTP response wrapping rendered HTML to the browser

How to define endpoint route in Flask

- Use `@app.route` or `blueprint.route`
- `<param_name>` for parameter called `param_name`
- `<int:param_name>` for enforcing parameter type
- Blueprint is used for grouping related requests/resources

The following converters are available:

<i>string</i>	accepts any text without a slash (the default)
<i>int</i>	accepts integers
<i>float</i>	like <i>int</i> but for floating point values
<i>path</i>	like the default but also accepts slashes
<i>any</i>	matches one of the items provided
<i>uuid</i>	accepts UUID strings

Custom converters can be defined using `flask.Flask.url_map`.

Here are some examples:

```
@app.route('/')
def index():
    pass

@app.route('/<username>')
def show_user(username):
    pass

@app.route('/post/<int:post_id>')
def show_post(post_id):
    pass
```

```
from flask import Blueprint, render_template, abort
from jinja2 import TemplateNotFound

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

@simple_page.route('/', defaults={'page': 'index'})
@simple_page.route('/<page>')
def show(page):
    try:
        return render_template(f'pages/{page}.html')
    except TemplateNotFound:
        abort(404)
```

What you need to implement end to end?

- Add new table in create.sql and load.sql for loading fake data
- Add new model under app/models/xxx.py (Example: app/models/user.py)
- Add new static method under the model class for each SQL query (Example: app/models/user.py::get (L92))
- Add new backend endpoint controller under app/xxx.py (Example: app/users.py::Login (L23))
- Add new HTML Jinja template under app/templates/xxx.html (Example: app/templates/login.html)