Monday: Templates; if Statements

Creating the Application.

Let us begin by creating our application. Create a new directory and call it *Watchlist*. Inside the project initialize git. Create a virtual environment inside our directory and name it virtual and activate it. This new directory should be on its own, not inside the *Hello World* directory we created earlier.

Application Structure

For the purpose of uniformity, we will be working with the following folder structure.

```
watchlist/
|
git
|
app/
    |-app/static/ # Stores all static files CSS,Javascript ,Images
    |
    |-app/templates/ # Stores all our template files.
    |
    |-app/__init__.py # Where we will initialize our application.
    |
    |-app/views.py # Where we will create all our view functions.
    |
    |-app/error.py # Where we will create handlers for error pages.
    |
    |-app/config.py # Where we will store our app configurations.
|
virtual/ # Our virtual Environment
|
.gitignore
|
README.md
|
run.py # File that will run our application
```

.gitignore

```
virtual/
*.pyc
```

Inside our *.gitignore* file we will exclude our virtual environment and also we will exclude all the compiled Python files `*.pyc`

Create First Template

Flask uses Jinja 2 as a **template language**. A template language is a simple format that is used to automate the creation of documents.

Let us first install flask into our virtual environment

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

Next, let us initialize our application.

`__init__.py`

```
from flask import Flask

# Initializing application
app = Flask(__name__)

from app import views
```

We import the `Flask` class from flask module and use it to create an app instance. We pass in the `__name__` variable.

We also import our `views` file from the *app* folder. This will allow us to create views.

Let us now create our index page view function. *views.py*

```
from flask import render_template
from app import app

# Views
@app.route('/')
def index():

    '''
    View root page function that returns the index page and its data
    '''
    return render_template('index.html')
```

We import the `render_template()` function from Flask. This function takes in the name of a template file as the first argument. It then automatically searches for the template file in our *app/templates/* sub directory and loads it.

We then import the `app` instance from the app folder.

We created a route decorator `@app.route('/')`, and defined the view function `index()`. As the response, we use the `render_template()` function and pass in `index.html` file which we will create.

Inside the *app/templates/* sub directory create the *index.html* file and add the following:

index.html

```
<h1> Hello World </h1>
```

Now let us run this application. Inside your root directory in the *run.py* file add the following code.

run.py

```
from app import app

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

Here we import the app instance. We then check if the script is run directly and then use the `app.run()` method to launch our server. We pass in the `debug = True` argument to allow us to run on debug mode so that we can easily track errors in our application.

Now we can run the application and see how it looks on our browser

```
(virtual)$ python3.6 run.py
```

Variables

In Jinja2, variables can be passed in and placed in predefined locations in our template file.

index.html

```
<h1> {{ message }} </h1>
```

We remove the text and add a **variable block** that will be replaced with dynamic content from flask. Variable blocks are defined with `{{}}` double curly braces and a variable name placed inside.

views.py

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

    '''
    View root page function that returns the index page and its data
    '''

    message = 'Hello World'
    return render_template('index.html',message = message)
```

We update our `index()` view function to add a `message` variable. We then pass the variable as an argument in our `render_template()` function. The first `message` on the left of the `=` sign, represents the variable in the template. While the one to the right represents the variable in our view function.

When we run the application we see the message displayed in our browser.

Dynamic routes

Some URLs might have dynamic sections in them. For example, consider `https://www.github.com/<username>` this loads a user's profile according to the username passed in.

Flask also supports this using a special syntax inside the route decorator.

views.py

```
@app.route('/movie/<movie_id>')
def movie(movie_id):

    '''
    View movie page function that returns the movie details page and its data
    '''
    return render_template('movie.html',id = movie_id)
```

We add a new route that has a `movie()` view function. The part in the angle brackets `<>` is dynamic. And any route mapped to this will be passed. This returns a response of a *movie.html* file.

By default, dynamic parts are rendered as strings but they can be transformed to be of any type

views.py

```
@app.route('/movie/<int:movie_id>')
def movie(movie_id):

    '''
    View movie page function that returns the movie details page and its data
    '''
    return render_template('movie.html',id = movie_id)
```

We changed the dynamic part to `<int:movie_id>` to transform content to integer.

templates/movie.html

```
<h1> Hello Movie {{ id }} </h1>
```

We run this application and in our browser URL we navigate to the following address. `http://127.0.0.1:5000/movie/1234`

We should see the `Hello Movie 1234` text displayed in our browser window.

Configurations

Our applications are going to need several configurations to fit our needs.

Creating Configuration classes

config.py

```
class Config:
    '''
    General configuration parent class
    '''
    pass



class ProdConfig(Config):
    '''
    Production  configuration child class

    Args:
        Config: The parent configuration class with General configuration settings
    '''
    pass


class DevConfig(Config):
    '''
    Development  configuration child class

    Args:
        Config: The parent configuration class with General configuration settings
    '''
    DEBUG = True
```

Here we created three classes. The parent `Config` class contains configurations that are used in both production and development stages. The `ProdConfig`subclass contains configurations that are used in production stages of our application and inherits from the parent `Config` class. The `DevConfig` subclass contains configurations that are used in development stages of our application and inherits from the parent `Config` class.

Inside `DevConfig` subclass, we add `DEBUG = True;` this enables debug mode in our application.

`__init__.py`

```
from flask import Flask
from .config import DevConfig

# Initializing application
app = Flask(__name__)

# Setting up configuration
app.config.from_object(DevConfig)

from app import views
```

To make our application use the new configurations, we import our `DevConfig `subclass in our `__init__.py` file. We then use `app.config.from_object()` method to set up configuration and pass in the `DevConfig` subclass.

run.py

```
from app import app

if __name__ == '__main__':
    app.run()
```

We now remove the `debug = True` argument from our `app.run()` because the debug mode has been enabled in the configuration file.

You can find the application at this point at https://github.com/mbuthiya/watchlist/tree/01-create-basic-template

Control Flows in Jinja templates

Jinja templates allow us to add some logic to our application.

views.py

```
def index():

    '''
    View root page function that returns the index page and its data
    '''

    title = 'Home - Welcome to The best Movie Review Website Online'
    return render_template('index.html', title = title)
```

We want to pass in a title to our templates instead of defining them inside the html structure. We create a variable `title` and pass it into our templates.

templates/index.html

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        {% if title %}
        <title> {{ title }}</title>
        {% else %}
        <title> Welcome to the best Movie Review website </title>
        {% endif %}
    </head>
    <body>

    </body>
</html>
```

We add HTML structure to our template. Inside our `<head></head>` we have a **control block** with an `if` statement.

Unlike the variable blocks defined with `{{}}` control blocks are defined using `{% %}`. They are used to declare functions,loops and if statements.

The `if` statement checks if we have provided a `title` variable for the template. In the chance that you forget to provide a `title` to your template the `else` control block is executed and a default title is used instead.

Task

For practice repeat the process we have gone through to the `movie.html` page displaying the movie id in the title.

You can find the application at this point here https://github.com/mbuthiya/watchlist/tree/02-control-flow-html-pages