15) Intro to backend web development

Agenda

- Website or web application?
- Front-end and back-end
- Flask
- MVC
- Routes
- Assets

In the **first part** of this course we learned about **HTML & CSS**.

In the **second part** we've learned about **the basic concepts of programming** with Python (variables, loops, data structures and functions).

Now we will **join** the knowledge from **both parts** and start building **web applications**!

Website or web application?

There are two types of websites:

- static websites
- dynamic websites

Static websites only have a front-end, which is built with HTML&CSS. But the problem is that this kind of website cannot do much more than just exist on the web.

It cannot accept input from users, like a **contact form** or **login** or other examples of forms.

If we want our website to be something more than just static HTML&CSS, it needs to have a **back-end.** A back-end makes website much more powerful. That's why we call these kind of websites **dynamic.** Another word for it is a **web application** (or web app).

When we say "app", we often think of a mobile app. But "app" stands for "application", which is a word used a lot longer than we have smartphones.

There are in fact three main types of software applications (or software apps):

- *desktop apps (or desktop programs, like the ones you created in the previous lessons)*
- *web apps (dynamic websites)*
- *mobile apps*

*"App" is just a synonym for a computer **program**.*

Front-end and back-end

Remember when we compared a web application to a car?

We said that the **car body** (or car shell) is like **front-end** and **car engine** is like **back-end.**

As we said, front-end is built with HTML&CSS. But for backend you can use basically any programming language: PHP, Java, Ruby, C#, ... or Python**.**

We will use Python, of course.

Flask

We will use a special Python pip package called **Flask** to help us build a web application.

Let's create a new Python project and put this into its **requirements.txt** file:

```
Flask==1.0.2
```

This line means that we will install a pip package named Flask with a version number "1.0.2".

Let's install it:

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

(Alternatively, you can install the package via PyCharm settings.)

Next create a new file called `main.py` and enter the following code in it:

```
from flask import Flask

app = Flask(__name__)


@app.route("/")
def index():
    return "Hello, SmartNinja!"


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

Now **right-click** the main.py file and select **Run**. A Python shell will open with this message:

```
 * Serving Flask app "main" (lazy loading)
 * Environment: production
   WARNING: Do not use the development server in a production environment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Click on the http://127.0.0.1:5000/ (http://127.0.0.1:5000/) link and your first web app will open in the Terminal.

Congrats! :)

Code explanation

Let's go step-by-step through the code in `main.py` in order to understand it.

First, we imported Flask into our Python file:

```
from flask import Flask
```

We've done that before with other Python libraries, like `random` and `datetime`.

Next we created a new Flask app object:

```
app = Flask(__name__)
```

We know what objects are, so this is nothing new.

But this part of the code is new:

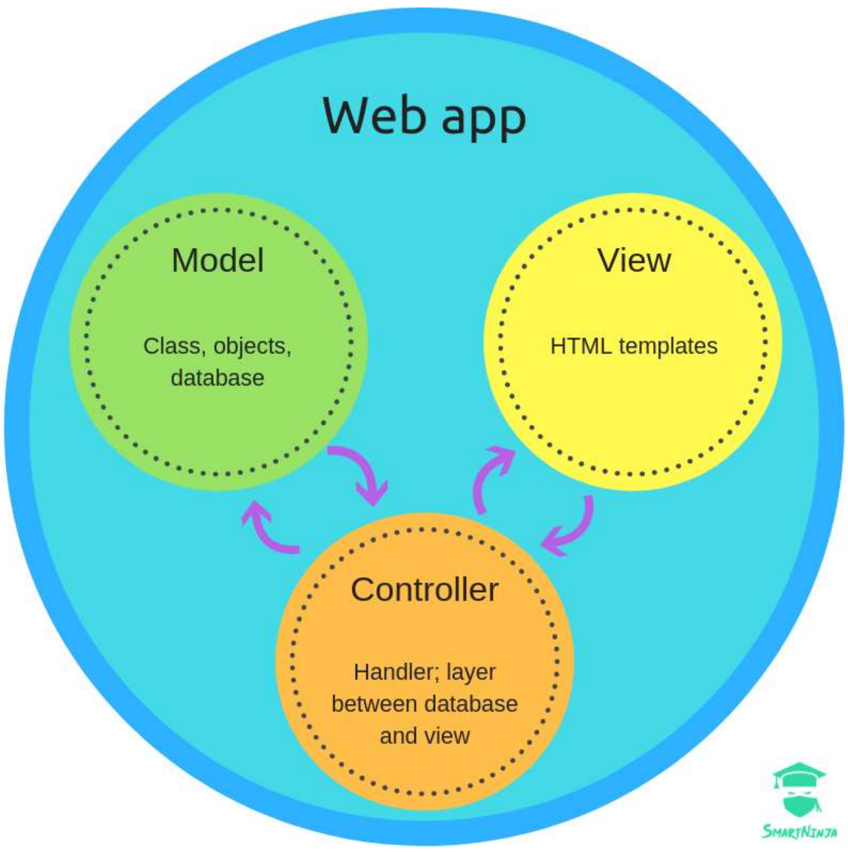
```
@app.route("/")
def index():
    return "Hello, SmartNinja!"
```

What is this?

This is a **controller**.

Model - View - Controller (MVC)

We mentioned some new term: **controller**. Modern web applications follow the concept called **Model - View - Controller (MVC)**.



We already know **model** from OOP. It helps us create new objects. In web applications which use a database, this is also a model for a database table.

View applies to HTML **templates** that our users see (a front-end).

Controller is a backend code that **controls** what users see in View and (with a help of Model) reads from and writes to the database. Controller is a sort of **middle layer** between the database and front-end.

> ***Important:*** *In Python we usually call controllers* ***"handlers"****. And we call views* ***"templates"****.*

The best way to understand this is by doing an example.

Controller-View connection

In this and the following couple of lessons we'll focus on a relationship between a controller (or a handler) and a View (HTML templates).

The other relationship (between Model and Controller) will be discussed in the Database lesson.

Okay, so currently we have a controller/handler already created:

```
@app.route("/")
def index():
    return "Hello, SmartNinja!"
```

Now let's create our View.

Create a **new folder** in your project and name it `templates` . Inside the templates folder create a new HTML file called `index.html` . Enter the following code in it:

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <title>Hello, SmartNinja</title>
    </head>

    <body>
        <h1>Welcome back to HTML ;)</h1>
    </body>
</html>
```

Ok, so now we have both controller and a view. The next step is to **connect them**.

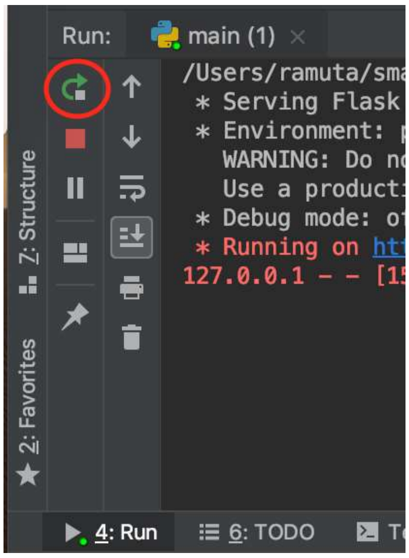
Go back to `main.py` and add `render_template` to your import:

```
from flask import Flask, render_template
```

Then change your handler a little:

```
@app.route("/")
def index():
    return render_template("index.html")
```

You will probably need to re-run your Python file in order to see the changes in your browser (click on the green arrow):



```
/Users/ramuta/sma
 * Serving Flask
 * Environment: p
   WARNING: Do no
   Use a product
 * Debug mode: of
 * Running on htt
127.0.0.1 - - [15
```

Now go back to your browser and reload the http://127.0.0.1:5000/ (http://127.0.0.1:5000/) page. You should see "Welcome back to HTML ;)".

Congrats, you have successfully connected your handler (controller) with your HTML template (view)!

Routes

Let's go back to examining our handler code:

```
@app.route("/")
def index():
    return render_template("index.html")
```

We can see it's a function called `index()` . And it returns the `index.html` template.

But what is this weird part above the function name:

```
@app.route("/")
```

This is the part that tells you at which URL this handler can be reached. Currently it's set to the **root** of the website: `/` .

This means that whenever we will go to `http://127.0.0.1:5000/` , this function will be called and it will return the `index.html` template.

What if we change the route to something else?

Let's change the URL route to `/hello` :

```
@app.route("/hello")
def index():
    return render_template("index.html")
```

Let's rerun the app in the Python shell and then go back to the browser and reload the http://127.0.0.1:5000/ (http://127.0.0.1:5000/) page.

What do we get?

```
Not Found

The requested URL was not found on the server. If you entered the URL manually please check you
```

Then go to this URL: http://127.0.0.1:5000/hello (http://127.0.0.1:5000/hello):

```
Welcome back to HTML ;)
```

We can see our HTML page again!

Short exercise (10 min)

> *Instructor can do this example together with students (instructor writing the code and students telling the instructor what to write).*

Let's create two more handlers and two more HTML files (so that we have three in total).

Change the `index()` handler URL back to `/`.

The other two handlers can be named:

- `about_me()`, with the `about.html` template and `/about` URL, and
- `portfolio()`, with the `portfolio.html` template and `/portfolio` URL.

Each HTML page should have links to the other two pages (don't use Jinja here yet).

Static files

So far we've only used plain HTML. But we know that we can make website front-end so much prettier with CSS. How can we add it to our website?

CSS files have to reside in a special folder called `static`. The same goes for all other **"static" files**, such as JavaScript files, fonts or images.

Let's create the `static` folder and within this folder create two folders: `css` and `img`.

CSS

Inside the `css` folder create a file called `style.css`. Add this code in it:

```
body {
    background-color: lightblue;
}
```

Now open the index.html file and add this line to access your CSS file:

```
<link rel="stylesheet" href="/static/css/style.css">
```

Re-run your `main.py` file again and refresh your browser tab. You should see a light blue background now.

Image

Next download this image (or find some other picture on Google):

(https://storage.googleapis.com/smartninja/smartninja-logo-1547584690.png)

Save this image as `smartninja-logo.png` into your static/img folder.

Now you can access this image in your HTML files using this code:

```
<img src="/static/img/smartninja-logo.png">
```

Reload the main.py and your browser. You should see this image now. Congrats!

Q&A

Any question?

Potential questions:

- What is "127.0.0.1"?
- What are web frameworks, like Flask or Django?

If there's enough time after Q&A, students can start working on their homework.

Homework 15.1: My homepage

Create a new Flask web app which will serve as your personal website. It should have the following pages:

- Index page (`/`), where you say hello to the visitors
- About me page (`/about`)
- Portfolio page (`/portfolio`): contains links to subpages with your front-end projects

The Portfolio page should have links to the HTML&CSS pages that you've created in the front-end part of this course:

- Fakebook (`/portfolio/fakebook`)
- Boogle (`/portfolio/boogle`)
- Hair salon (`/portfolio/hair-salon`)

Create a separate handler (and URL route) for each of these pages.

When you finish, push your code to GitHub and paste the link to it on the forum.

Homework 15.2: Hosting providers

In the lesson you'll learn how to push your code to three different hosting providers. Create an account on each of them:

- Google Cloud (https://console.cloud.google.com/) (you need a Gmail account)
- Heroku (https://www.heroku.com/)
- Microsoft Azure (https://azure.microsoft.com/en-us/free/)

On some of these you might be asked to enter your credit card info (for verification purposes). We will only use free services on each of these platforms, so you can enter the credit card info without fearing you'll be charged.