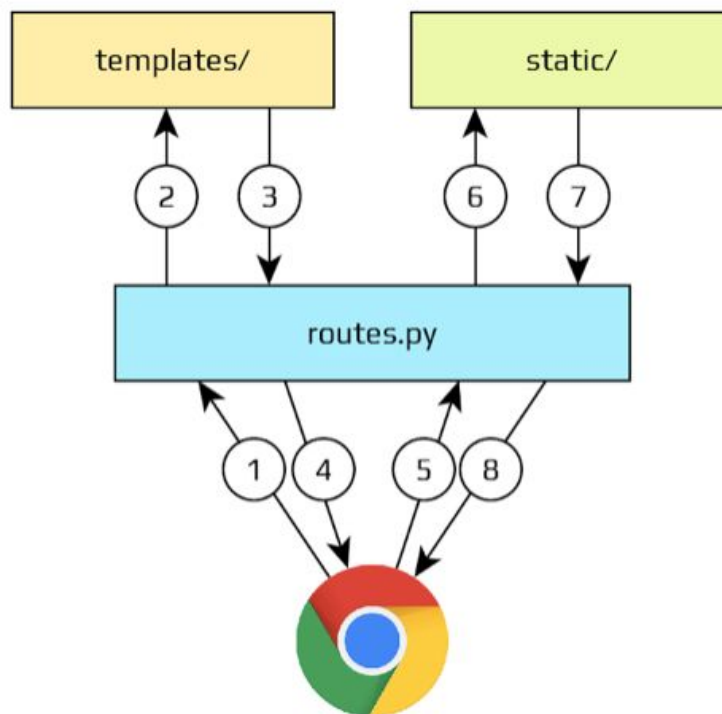# E-14a
Lab 2: The Request-Response cycle

*The Request-Response Cycle*

Learning Objectives

- Understand Request-Response cycle
- Create Flask app top-to-bottom
- Understand Jinja templates
- Working with Bootstrap framework
- Deploy using Heroku

INTRODUCTION

Last week, at the end of the introductory Lab we run our first Flask app. In this Lab we will cover more in-depth Flask and understand building blocks. Our task is to create the app that will consist of a home page with mostly static content. At the end, we will have a workflow that we can generalize to build more complex and dynamic pages.

Setting up the folders structure

Let's start with setting up the files and folders needed for the app. Let's start by creating a couple folders and files to keep the web app organized. Feel free to use a folder structure from a previous *lab_1*.
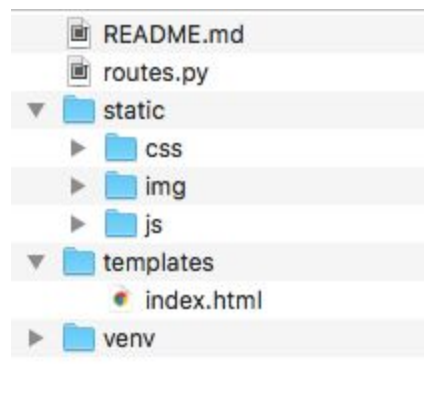

App's files and folders structure

Rename your *lab_1* folder into *lab_2* to avoid any further confusion. As a first step let's get back into the isolated development environment we created when we installed Flask. Open the terminal and navigate to your *lab_2* folder. Now you have to tell the system that you want to use your virtual environment. To activate your virtual environment use the following command lines:

```
$ source venv/bin/activate
(venv) $ _
```

Before opening *routes.py* file and start building the app, let us trace what happens when a user visits the page in the Flask app.

The Request-Response cycle

In order to put all of the above files and folders together we need to understand the **request-response cycle**, or what happens when a user visits a Web page. The cycle starts with opening the browser, typing a URL and pressing enter. After that, the browser sends a **request** (by using the HTTP protocol) to the web server. After receiving the request, Web server loads requested resource and sends it back as a **response**. After receiving the response from server, browser interprets it and displays the result.
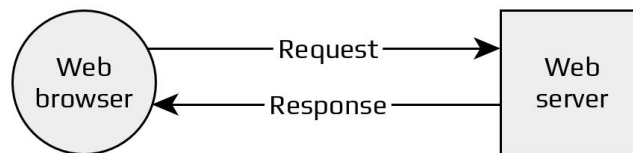
However, modern Web pages are usually more complex than just a bunch of default formatted text - they contain images, specific formatting styles, and even some interaction (written in JavaScript). All these pieces are linked in basic Web page, and are (in most cases) loaded within separated request-response cycles (that's why sometimes web page loads, but font is changed after few seconds.). It's not unusual that for loading a single Web page more than 50 request-response cycles are needed (though too many requests have a bad impact on loading performance).
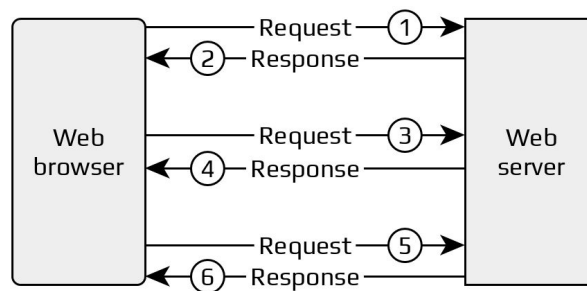
Most of the modern Web browsers have integrated developer tools, that you could use to check how many requests are needed to load some web page:

Name	Status	Type	Initiator	Size	Time	Waterfall
localhost	200	document	Other	304 B	9 ms	
main.css	200	text/css	(index)	28 B	15 ms	

Web server is just a computer program (written in Python, C, or any other programming language that listens on a certain port (80 and 443 are default), parses received requests,

loads requested content, and sends it as a response. This is a very simple task when serving **static** content (like HTML pages, images, CSS and JavaScript files):
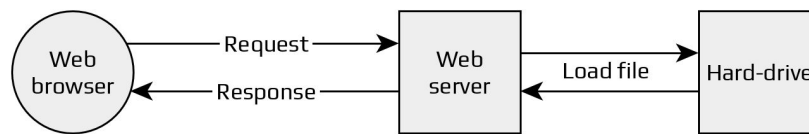

But, when serving **dynamic** content, things become a bit more complicated. The dynamic content is not already available on a hard-drive, in its final form, but it is rather generated for each individual request. This content generation is done by executing a certain program. For example, if you want to include some data from a database in your Web page, you'll have to write a program that connects to the database, loads some records from a table, and formats the result as a HTML page. After that, that HTML content (response) is sent back to the Web browser in the same way as a previously mentioned static page.
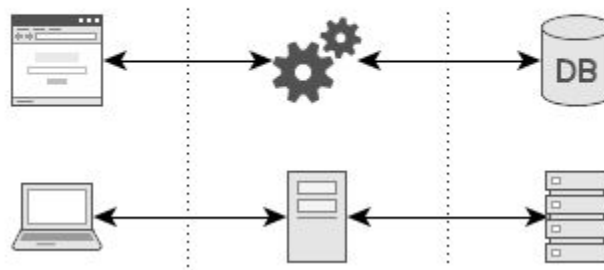
Static content like images, css or javascript can be delivered with a simpler servers (that only loads and sends the contents of the file rather than processing the additional requests). When application has many users, then it's common practice to move these static files to the *Content Delivery Networks* (e.g. "Cloudfare" instance) scattered all over the world to serve users from a various geographic locations.

So, let's go back to our Flask application. Flask is a micro-framework that includes a Web Server Gateway Interface (WSGI), which means that it acts as a Web server, and a templating engine (Jinja), which makes it easier for us to create dynamic content. The program we wrote (routes.py) import Flask, and starts a simple Web server that parses requests, loads requested content, and sends it back as a response.
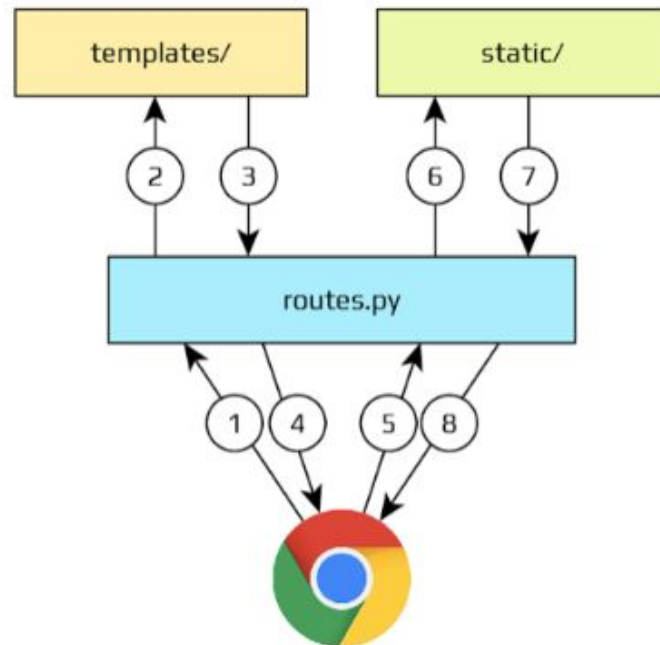
There are two different types of requests for our Web server: the requests for the dynamic content (templates, stored in the folder 'templates/'), and the requests for static files (images and CSS files, stored in the folder 'static'). These static files are linked in our templates, for example:

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

So, when we execute our program 'routes.py', and when we type URL 'http://127.0.0.1:5000/' in our browser, following steps are taken:

1. Web browser sends a **request** to the Web server (program 'routes.py', executed on the local computer) - (step 1).
2. Web server (Flask) loads and executes (interpretes) appropriate template from the 'templates/' folder - (steps 2 and 3).
3. Web server sends the result as a response to our Web browser - (step 4).
4. After receiving the response, the Web browser parses and interpretes received HTML code.
5. As this code contains links to various static resources (images, css...), a separate request (step 5) is sent to the Web server.
6. Web server loads the requested static resource (folder 'static/', steps 6 and 7), and sends it back to the Web browser (step 8).
7. Finally, after receiving the requested resource, Web browser interpretes it and includes it in the Web page. At this point we can consider the requested page to be

completely loaded.

Working with Templates

Templates are a useful technique that allows you to concentrate on one part of the application without having to worry about other parts of the system that don't exist yet. As we learned in Lectures, we want to design with a possibility to adapt to users. In that case we need to be able to modify certain aspects of our app keeping the basic layout unchanged.

Templates are html files that contain variables and control close statements. Rather than writing the same html over and over again in multiple web pages, we write the common html once in a base template, and inherit it from the child pages. Suppose we want to make another page (or two). Our app should have a similar style and layout across all of its pages, so that means copy/pasting all the HTML content from the *index.html* over and over again. It seems pretty wasteful to write all of this HTML again, especially when the construction of the page will remain the same.

Templates help achieve the separation between presentation and background logic. In Flask, templates are written as separate files, stored in a separate folder. Below, you can see a very simple template structure:

```
<html>
    <head>
        <title>{{ title }} </title>
    </head>
    <body>
        <h1>Hello, {{ user.username }}!</h1>
    </body>
</html>
```

Let us use this logic to create templates for our app. Open your terminal and navigate to *lab_2/templates.* Let us begin by making a new file in templates called *layout.html.* The output should look like this:

```
[(venv) Z-Mac:templates zonakostic$ touch layout.html
[(venv) Z-Mac:templates zonakostic$ ls
 index.html        layout.html
 (venv) Z-Mac:templates zonakostic$ ▊
```

We'll call *layout.html* the base template. Here's what *layout.html* should look like:

```html
<html>
  <body>
    <header>
      <div class="container">
        <h1 class="title">Hello World</h1>
      </div>
    </header>

    {% block content %}
    {% endblock %}


  </body>
</html>
```

It looks just like what we had in *index.html* before, except now there is this **block content endblock** part. Now, go to the *index.html* and and replace the previous content with this:
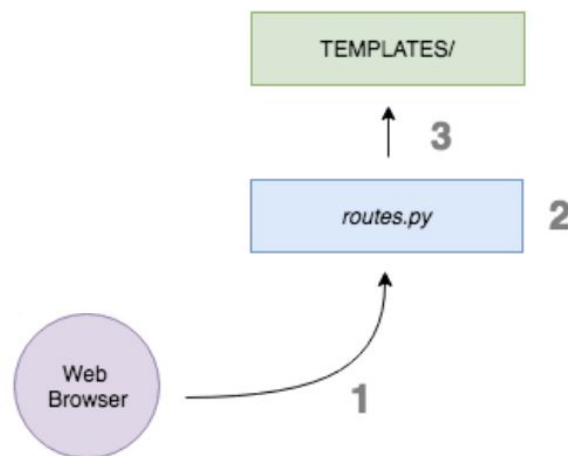
```html
{% extends "layout.html" %}
{% block content %}
  </main>
      <div class="section-content">
        <h2>Index Block Content</h2>
      </div>
  </main>
```

```html
{% endblock %}
```
In this way, the base template defines the common elements of the website, while the child elements customize it with their own content. This is a mostly standard, very simple HTML page. The only difference is in a couple of placeholders for the *dynamic content*, enclosed in `{{ ... }}` sections. These placeholders represent the parts of the page that are variable and will only be known at runtime.

Routing

We just created a template and we placed it inside the templates folder. For us to see this page in the browser, we need to map a URL to it, and we do this in *routes.py*. Once you have the page loaded in your browser, you may want to view the source HTML and compare it against the original template.

The operation that converts a template into a complete HTML page is called **rendering**. To render the template we had to import a function that comes with the Flask framework called `render_template()`. This function takes a template filename and a variable list of template arguments and returns the same template, but with all the placeholders in it replaced with actual values.

The `render_template()` function invokes the [Jinja2](#) template engine that comes bundled with the Flask framework. Jinja2 substitutes `{{ ... }}` blocks with the corresponding values, given by the arguments provided in the `render_template()` call.

Let us take a look at *routs.py* code[1]. Use `pwd` to locate your current directory. If you are not into your *lab_2* directory, navigate back and open up *routes.py*, and modify it following next steps. First, import the Flask class, and the function `render_template`:

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

---

[1] Python OOP: https://python.swaroopch.com/oop.html

Then create a new usable instance of the Flask class and save it into the variable app:

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

Third, map the URL "/" to the Python function index:
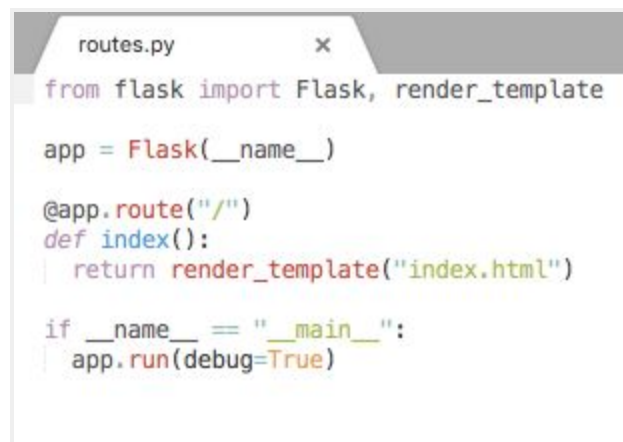
```
@app.route("/")
```

The Python function uses the Flask function render_template to render index.html.

```
def index():
   return render_template("index.html")
```

So now when a user types in the URL "/", the function index will run and return the page index.html. Finally, finish up the file with this if statement, app.run runs the app on a local server, the debug=True flag here is set so that we'll see any error messages along the way.

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

The final code should look like this:

Sublime output of routes.py

Make sure you save your *routes.py* file. Go to the command line and type `python routes.py` to start the local server. Go to your browser and type `localhost:5000` in the address bar.

HTML, CSS & JS Frameworks

Rather than coding from scratch, frameworks enable you to utilize ready made blocks of code to help you get started. They give you a solid foundation for what a typical web project requires and usually they are also flexible enough for customization.

Before we start with Bootstrap, let us add more HTML elements. Before moving forward, please make sure your *index.html* and *layout.html* look exactly like this:

layout.html

```
<html>
 <body>

     <!-- Navigation →
  <nav class="navbar navbar-expand-lg navbar-light fixed-top" id="mainNav">
      <div class="container">
        <div class="collapse navbar-collapse" id="navbarResponsive">
          <ul class="navbar-nav ml-auto">
            <li class="nav-item">
              <a class="nav-link" href="index.html">Home</a>
            </li>
            <li class="nav-item">
              <a class="nav-link" href="about.html">About</a>
            </li>
            <li class="nav-item">
              <a class="nav-link" href="contact.html">Contact</a>
            </li>
          </ul>
        </div>
      </div>
    </nav>

    {% block content %}
    {% endblock %}

  </body>
</html>
```

index.html

```
{% extends "layout.html" %}
{% block content %}


  <!-- Page Header -->
<header class="masthead"
style="background-image: url('static/img/home-bg.jpg')">
        <div class="overlay"></div>
        <div class="container">
          <div class="row">
            <div class="col-lg-8 col-md-10 mx-auto">
              <div class="site-heading">
                <h1>Clean Blog</h1>
             <span class="subheading">A Blog Theme by Start Bootstrap</span>
              </div>
            </div>
          </div>
        </div>
      </header>


      <!-- Footer -->
      <footer>
        <div class="container">
      <p class="copyright text-muted">Copyright &copy; Your Website 2018</p>
        </div>
      </footer>

{% endblock %}
```

In this Lab we will be working with [Bootstrap](#) as an example open source HTML, JS and CSS framework. It is one of the most widely used frameworks, it is easy to understand and it provides a great documentation with many examples. Here is a summary of the main aspects of Bootstrap:

- Open source HTML, CSS, and JS framework
- Provides a base styling for common used HTML elements
- The grid system helps you to create multi-column and nested layouts
- Extensive list of pre-styled components
- Customizable: All CSS rules can be overridden by your own rules
- Compatible with the latest versions of all major browsers

Download Bootstrap, Include the Bootstrap files in your *index.htm*. We want to use the CSS file across all the web pages. This means that we need to add the CSS file to the base template *layout.html*. Open up *layout.html*, and add this two lines inside the head element:

```
<head>

<link    href="{{    url_for('static',    filename='css/bootstrap.min.css')    }}"
rel="stylesheet">

<!-- Custom styles for this template -->
<link href="{{ url_for('static', filename='css/clean.css') }}" rel="stylesheet">

</head>
```

We are using Flask function URL4 to generate a URL for us. URL4 is telling Flask to go to the static folder and look for the file `bootstrap.min.css`.

Before running your app make sure you have necessary files included! Download *clean.css* from here and *home-bg.img* from here. Put CSS file into *static/css* folder and image into *static/img*.

Save your changes, go back to your browser, and reload the page. The CSS should now be

applied, and your *index.html* page should look the same. Go to the official Bootstrap website and skim over the different styles and configured **components**. Take a look at some grid examples of Bootstrap.

Publishing and Saving

GitHub

The workflow starts on your computer. You're using a text editor to write your code, and you're using the command line to run your code. Now, as you write more code you should keep track of it using Git and and store that code on GitHub. When the application is ready to share with others, we should deploy Heroku, so that users can go to that app's URL and view the site. So far, we've finished step one, by making a homepage, so let's proceed with step two and push the code up to GitHub.

To begin, go to the terminal and shut down the local server by typing `Control + C`. We'll restart this later when we've added new features. Create a new Git repository. Type git init inside your application. Let's verify that the new Git repository was created. Type `ls -al`. This command lets you see all files, even hidden ones. And here we see that the *.git* folder was created, so a new repository was created successfully. Next, let's add the files we changed to this repo:

```
git init
ls -al
git status
git add -A
git commit -m "Initial commit"
```

Before we push our files, go to your Git page and open your new repo. After creating new repo you should take a look at this section of the screen (you will have the same set of commands, but with your username and repo name):

...or push an existing repository from the command line

```
git remote add origin https://github.com/zonakostic/lab02.git
git push -u origin master
```

First line connects your local repo to a git repo. second line pushes all files to a git repo.

Refresh the page and you will be able to see your commits.

```
git remote add origin
        https://github.com/zonakostic/lab02.git
git push -u origin master
```

You might be asked for your credentials. Enter your username and password and continue with uploading your files to the Git repo.

For more information about Git commands, check this page: https://git-scm.com/book/en/v2

Heroku

To deploy it to Heroku, we have to do three things. First, install the Gunicorn web server. Type `pip install gunicorn`.

Second, create a file named `requirements.txt` that has a list of the Python libraries we have installed so far. Type `pip freeze > requirements.txt`. And here's the file that was created.

```
[(venv) Z-Mac:lab2_files zonakostic$ pip freeze > requirements.txt
(venv) Z-Mac:lab2_files zonakostic$ █
```

```
📄 README.md
📄 requirements.txt          click==6.7
📄 routes.py                 Flask==1.0.2
📁 static              ▶     gunicorn==19.9.0
                             itsdangerous==0.24
📁 templates          ▶     Jinja2==2.10
📁 venv               ▶     MarkupSafe==1.0
                             Werkzeug==0.14.1
```

Third, add a text file named Procfile that tells Heroku to run Flask using Gunicorn. Then in Sublime add this line to the *Procfile*.

```
web: gunicorn routes:app
```

Now that we've done these three things, let's create a new Heroku App. In the terminal, type `heroku login` then `heroku create`. This will create a new sub-domain for your app. The Heroku create command creates a new Git remote repository on the Heroku servers. We can push the Flask app to this remote repository using git push heroku faster, but before we do this let's make sure to commit the new files we added, *Procfile* and *requirements.txt*.

Once the new files are committed, go ahead and deploy the app using `git push heroku master`. The app will take a few seconds to deploy. See what it looks like by typing `heroku open`. Heroku open will open a new tab for you and this app is live for anyone to see. You can copy this URL and share it with others, so that they can see your progress so far.

```
[(venv) Z-Mac:lab2_files zonakostic$ heroku login
heroku: Enter your login credentials
Email: zonakostic@g.harvard.edu
[Password: ************
Logged in as zonakostic@g.harvard.edu
[(venv) Z-Mac:lab2_files zonakostic$ heroku create
Creating app... done, ● immense-escarpment-66821
https://immense-escarpment-66821.herokuapp.com/ | https://git.heroku.com/immense
-escarpment-66821.git
(venv) Z-Mac:lab2_files zonakostic$ █
```

As a final step, make sure to push your changes to the GitHub as well. Type `git push origin master`.

zonakostic Initial commit		Latest commit 8d7cd9d 10 minutes ago
static/css	Initial commit	10 minutes ago
templates	Initial commit	10 minutes ago
venv	Initial commit	10 minutes ago
Procfile	Initial commit	10 minutes ago
README.md	Initial commit	10 minutes ago
requirements.txt	Initial commit	10 minutes ago
routes.py	Initial commit	10 minutes ago

Take home exercises

Create About and Contact pages using files from *lab_2*.
https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-ii-templates

Flask Progressive Apps

https://github.com/umluizlima/flask-pwa
https://developers.google.com/web/fundamentals/codelabs/your-first-pwapp/
https://developers.google.com/web/fundamentals/primers/service-workers/
https://github.com/umluizlima/flask-pwa

Credits and Additional Resources

The Flask Mega Tutorial book:
https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-i-hello-world
https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-ii-templates

Jinja:
http://jinja.pocoo.org/

Flask Web Development - Developing Web Applications with Python:
https://flaskbook.com/

Flask with Bootstrap and Jinja Templating
https://pythonprogramming.net/bootstrap-jinja-templates-flask/
Flask Request-Response Cycle (Python example):

http://www.wellho.net/resources/ex.php4?item=y307/rrc

Python OOP:

https://python.swaroopch.com/oop.html