Image Classifier App - Part 2

Instructors **Battista Biggio**, **Angelo Sotgiu** and **Leonardo Regano**

M.Sc. in Computer Engineering, Cybersecurity and Artificial Intelligence
University of Cagliari, Italy

Thanks to Maura Pintor for kindly allowing the reuse of her material.

Part 2: Getting started with the code

Download the repository (run a terminal in the directory where you want to download it, or `cd` into that from your home directory):

```
git clone https://github.com/unica-isde/web-server-2023
```

Optional but recommended - create conda environment:

https://docs.conda.io/projects/miniconda/en/latest/

```
conda create --name isde python=3.12
conda activate isde
```

Let's explore the code repository.

It's a good practice to start from the `Readme.md` file and the `requirements.txt`. These are files that describe what the repository is for, and what is needed to run it.

The requirements file is like a shopping list. We can install all the libraries we need by typing:

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

This line will install the required libraries in your conda environment.

Follow the remaining steps in the Readme.

Open the folders and files in the project and familiarize with them.

Explore the `app` directory. We will just go through the main components.
This is what happens in collaborative projects. You have to understand the code just enough to contribute where it is needed.

Implement the first API

In the `main.py` file:

```python
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
def read_root() -> dict:
    """Returns the hello world first page."""
    return {"Hello": "World"}
```

First, we will try and run the server locally. We can just run the command:

```
fastapi dev
```

or alternatively:

```
uvicorn main:app --reload
```

We read on the terminal:

```
Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
```

This is a simple Python server running **locally** on our computer. This means that there is a service that is listening in the localhost address (127.0.0.1), port 8000, waiting for HTTP requests.

Now try to connect to the url from your web browser:
http://127.0.0.1:8000/

What happens when you click to the URL?

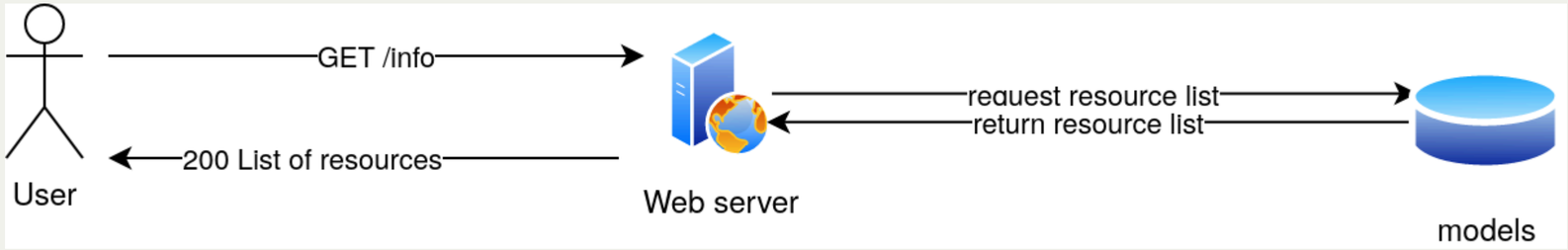
The browser is issuing a GET request to the server (look at your terminal), and the server is returning a python dictionary that will be encoded as JSON automatically by FastAPI. Finally, the browser renders the JSON as text.

What is good about FastAPI is that it creates the docs automatically:
http://127.0.0.1:8000/docs

Check out all the description that is automatically filled in by the framework.
Take time also to test the API with the GUI.

Now let's implement our first API!

We will keep it simple and just store a list of all models and images available in our server.



```python
from app.utils import list_images
from app.config import Configuration

@app.get("/info")
def get_info() -> dict[str, list[str]]:
    """Returns a dictionary with the list of models and
    the list of available image files."""
    list_of_images = list_images()
    list_of_models = Configuration.models
    data = {
        "models": list_of_models,
        "images": list_of_images
    }
    return data
```

Then try to navigate to http://127.0.0.1:8000/info

It's ugly, right[1]? It's because our web browser is usually rendering HTML files rather than raw JSONS.

---

[1] except for Firefox users, that will actually see an output formatted better than the others - but this is just an extra service offered by the browser

Let's instead reply to our request with an HTML file.

```python
from fastapi import FastAPI, Request
from fastapi.templating import Jinja2Templates
from fastapi.responses import HTMLResponse
from fastapi.staticfiles import StaticFiles
from app.utils import list_images
from app.config import Configuration

app.mount("/static", StaticFiles(directory="app/static"), name="static")
templates = Jinja2Templates(directory="app/templates")

@app.get("/", response_class=HTMLResponse)
def home(request: Request):
    return templates.TemplateResponse(request, "home_simple.html")
```

Navigate to http://127.0.0.1:8000/

Also, if you have some time, check out the HTML files that are in the templates directory.

Now we can finally implement our functionality. Let's ignore for now the fact that we have to build a queue.

We want the user to be able to insert data in a special construct, that will be used to gather the input for our service.

We should get the classification output with our machine learning utilities:

```
classification_scores = classify_image(model_id=model_id, img_id=image_id)
```

What can the user select?

- the image
- the machine learning (pretrained) model

We are going to use a form. Forms can be very customizable, and need often validation strategies to prevent users to insert uncontrolled inputs[2].

---

[2] **REMEMBER THIS WHEN YOU WORK ON THE PROJECTS**

Let's have a look at the HTML in `templates/classification_select.html.`

Note the images and models keys. These should be passed by Python from the backend.

```python
@app.get("/classifications", response_class=HTMLResponse)
def create_classify(request: Request):
    return templates.TemplateResponse(
      request, "classification_select.html",
      {"images": list_images(), "models": Configuration.models}
    )
```

Let's now see `templates/classification_output.html`

This file will display the scores that are passed (by the backend, again) into the variable `data['classification_scores']`.

```python
from typing import Annotated
from fastapi import FastAPI, Request, Form
from fastapi.templating import Jinja2Templates
from fastapi.responses import HTMLResponse
from fastapi.staticfiles import StaticFiles
from app.utils import list_images
from app.config import Configuration
from app.ml.classification_utils import classify_image

@app.post("/classifications", response_class=HTMLResponse)
async def request_classification(
    request: Request,
    model_id: Annotated[str, Form()],
    image_id: Annotated[str, Form()]
):
    classification_scores = classify_image(
      model_id=model_id,
      img_id=image_id
    )
    data = {
        "model_id": model_id,
        "image_id": image_id,
        "classification_scores": classification_scores
    }
    return templates.TemplateResponse(
      request, "classification_output.html", {"data": data}
    )
```

Change the html file in the home function to return the `home.html` file instead of `home_simple.html`

```
1 def home(request: Request):
2   templates.TemplateResponse(request, "home.html")
```

Try out the service now. Go to http://127.0.0.1:8000/ and navigate to the classification service. Pick an image and a model and see the results.

If we click on submit, the classification output should appear in our browser as a table with the top 5 scores.

We won't inspect the front-end in detail, but remember that we created a **form object** that is passed through the **FastAPI APIs** to the HTML file we are rendering through our browser.

What happens if we get many requests? What happens if the classification takes too long to process?

What happens if we get many requests? What happens if the classification takes too long to process?

If we don't send a response to users in a short time, they can get bored with our service, or worse, send more requests!

We can simulate a long running task by adding a line in the classification function.

```
import time

time.sleep(5)
```

The solution: implement a task queue.

Whenever the user sends a request, the server returns a status code. The web browser then can request the resource after a certain amount of time, and check the status of the queue.

This pattern is called polling, and is a mechanism that allows Asynchronous long running operations with the REST APIs.

First, we have to create a queue. We can do so in our classifications handler, and enqueue the jobs as soon as they are requested by users.

Let's edit our classification API

```python
from redis import Redis
from rq import Queue

...

@app.post("/classifications", response_class=HTMLResponse)
async def request_classification(
    request: Request,
    model_id: Annotated[str, Form()],
    image_id: Annotated[str, Form()]
):
    redis_conn = Redis(Configuration.REDIS_HOST, Configuration.REDIS_PORT)
    q = Queue(name=Configuration.QUEUE, connection=redis_conn)
    task = q.enqueue(classify_image, model_id=model_id, img_id=image_id)
    return templates.TemplateResponse(
        request, "classification_output_queue.html",
        {"image_id": image_id, "jobID": task.id}
    )
```

Notice that the HTML form that we are using has a `<script>` tag, which is running a `JavaScript` fragment. We are not going to edit that, but I will tell you what it is going on...

The script is run at the first time when the HTML is rendered.
Inside that, we have a polling mechanism that keeps asking for the resource `/classifications/{JobID}` every second, until the output JSON of the API says `"status": "success"`.

Now we should return that status and eventually the job result in a new API called `classifications/{JobID}`.

```python
@app.get("/classifications/{job_id}")
def classifications_id(job_id: str):
    redis_conn = Redis(Configuration.REDIS_HOST, Configuration.REDIS_PORT)
    q = Queue(name=Configuration.QUEUE, connection=redis_conn)
    task = q.fetch_job(job_id)
    print(task.return_value())
    response = {
        "task_status": task.get_status(),
        "data": task.return_value(),
    }
    return response
```

Now, we should run the worker and the server together.
See also the output that they produce.

What is happening (1/3):

What is happening (1/3):

- **frontend (html + javascript)**: the user requests the webpage.

What is happening (1/3):

- **frontend (html + javascript)**: the user requests the webpage.
- **backend(python)**: the server returns the html with the image and model selection.

What is happening (1/3):

- **frontend (html + javascript)**: the user requests the webpage.
- **backend(python)**: the server returns the html with the image and model selection.
- **frontend (html + javascript)**: the user picks the model and the image. The web browser issues the request to the backend server.

What is happening (2/3):

What is happening (2/3):

- **backend(python)**: the server receives the request and puts the task in the queue. Returns the id of the stored job to the browser and redirects to the results page.

What is happening (2/3):

- **backend(python)**: the server receives the request and puts the task in the queue. Returns the id of the stored job to the browser and redirects to the results page.
- **frontend (html + javascript)**: the web browser renders the result page and asks for the job result. If the status of the job is "success", the server renders the resulting output, otherwise it waits and repeat.

What is happening (3/3):

In the meanwhile...

What is happening (3/3):

In the meanwhile...

- **the worker (python)**: the worker takes the tasks from the queue and processes them, storing the result in the database.

This service *works*, but of course this is not the only requirement.

Depending on the application, we have always to add the "implicit" requirements like security, stability and documentation[3].

---

[3] not covered in this lesson, but always keep them in mind!