

Web Development

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.

What this lesson covers:

- designing APIs
- inspecting code written by others
- implementing a few basic APIs
- creating a container
- creating an architecture with isolated components
- scaling

Let's imagine a scenario:

We are a team of web developers that should build a demo of the product "image classifier", which is provided by another team of our company.

The code for the classifier is already written as it is a product of our company, we are going to use it as a **black box**.



We are not going to start from scratch.

The team has a repository that contains already some code.

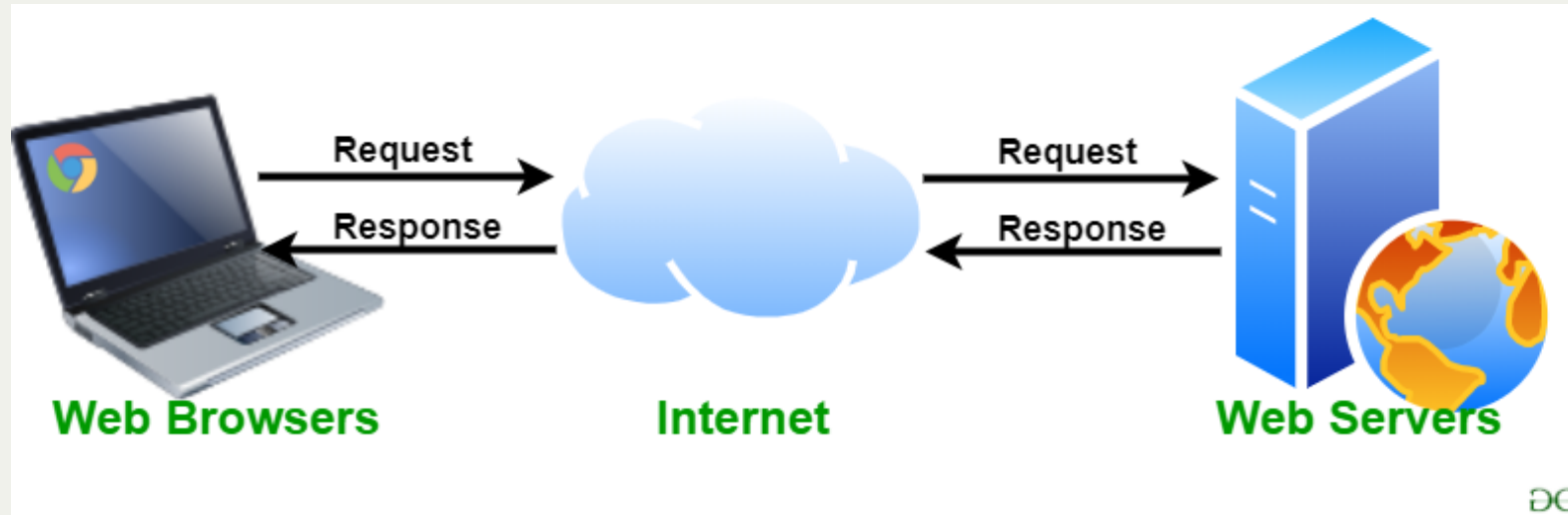
But first... let's see some fundamentals of web development

Part 0: Basics

Web servers, image classification, and containers

Web servers

Web server for the user



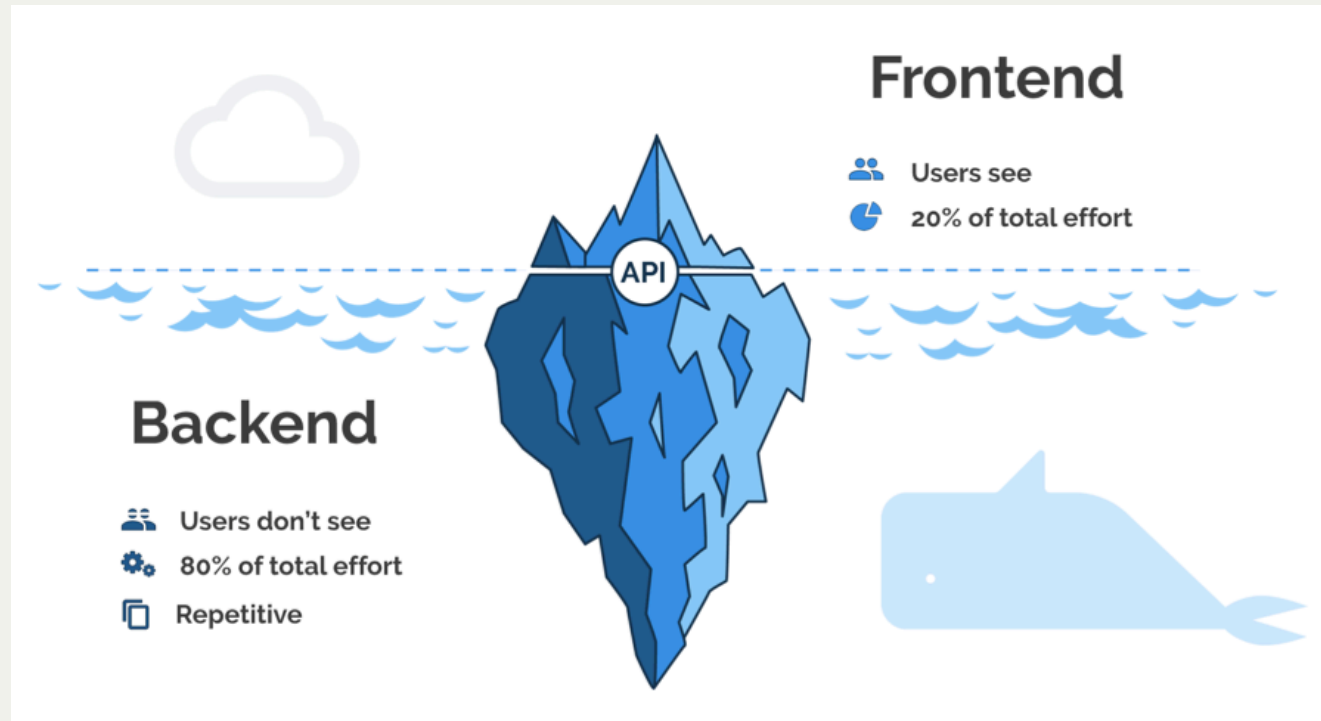
More info here.

Example of webserver

A web server provides resources (webpages, files, etc.) to clients (browsers, applications, etc.) through the Internet.

Clients send *requests* to web servers and receive *responses* from them.

Web server for the developer



More info here.

API

What is an API?



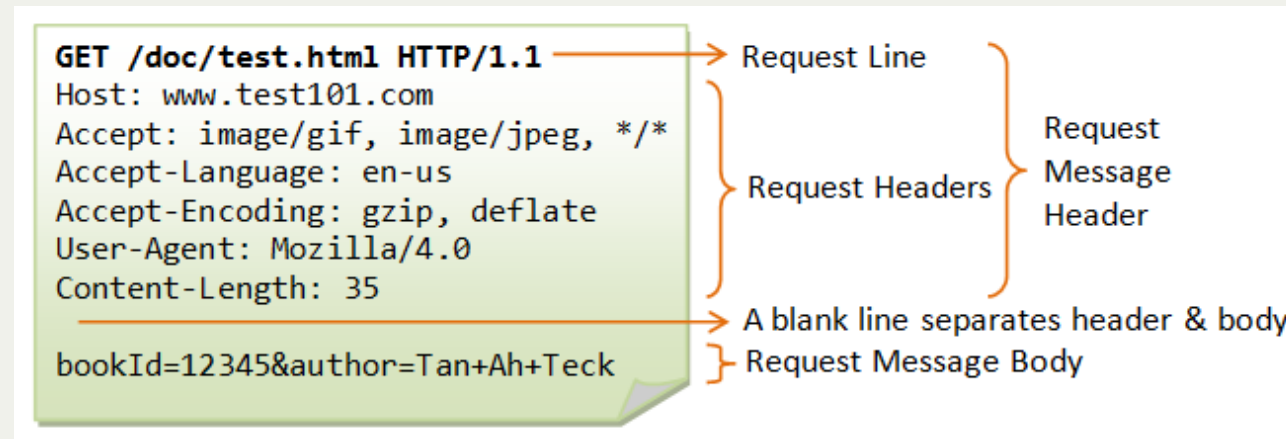
Open video

More info about APIs.

The language of web servers

GET	retrieve information
HEAD	retrieve resource headers
POST	submit data to the server.
PUT	save an object at the location
DELETE	delete the object at the location

HTTP



More info on HTTP Protocol.


HTTP

```
HTTP/1.1 200 OK
Date: Sun, 08 Feb xxxx 01:11:12 GMT
Server: Apache/1.3.29 (Win32)
Last-Modified: Sat, 07 Feb xxxx
ETag: "0-23-4024c3a5"
Accept-Ranges: bytes
Content-Length: 35
Connection: close
Content-Type: text/html

<h1>My Home page</h1>
```

Labels and structure:

- Status Line:** Points to the first line: `HTTP/1.1 200 OK`
- Response Headers:** A bracket groups the lines from `Date:` to `Content-Type: text/html`.
- Response Message Header:** A bracket groups the **Status Line** and the **Response Headers**.
- A blank line separates header & body:** Points to the empty line between the headers and the body.
- Response Message Body:** A bracket groups the body content: `<h1>My Home page</h1>`



WIKIPEDIA
The Free Encyclopedia

[Main page](#)
[Contents](#)
[Current events](#)
[Random article](#)
[About Wikipedia](#)

Article

Talk

Read

Edit

View history

Search Wikipedia

Machine learning

From Wikipedia, the free encyclopedia

For the journal, see [Machine Learning \(journal\)](#).
"Statistical learning" redirects here. For statistical learning in linguistics, see [statistical learning in language acquisition](#).

Machine learning (ML) is the study of computer algorithms that improve automatically through experience.^[1] It is seen as a subset of [artificial intelligence](#). Machine learning algorithms build a model based on sample data, known

Part of a series on
Machine learning

26

1.70 MB

1.32s

1

1

0

Search

Network

Storage

Console

Resources

Timelines

Events

Frames

Timeline Recording 2

Network Requests

0ms – 15.00s

Edit

1000.0ms

2.00s

3.00s

4.00s

5.00s

6.00s

7.00s

8.00s

9.00s

10.00s

Network Requests

Layout & Rendering

JavaScript & Events

Details

Images

Filter

Name	Domain	Type	Meth...	Sche...	Status	Cached	Size	Transferred	Start Time ^	Latency	Duration	2.00s
220px-Fig-X_Al...	upload.wikime...	Image	GET	HTTPS	200	Yes (Mem...	8.57 KB	0 B	711.0ms	0.127ms	0.056ms	
220px-Fig-y_Pa...	upload.wikime...	Image	GET	HTTPS	200	Yes (Mem...	5.76 KB	0 B	711.5ms	0.119ms	0.053ms	
220px-Svm_ma...	upload.wikime...	Image	GET	HTTPS	200	Yes (Mem...	9.31 KB	0 B	742.3ms	0.225ms	0.076ms	
2e6daa2c8e55...	wikimedia.org	Image	GET	HTTPS	200	Yes (Mem...	10.29 KB	0 B	744.5ms	0.196ms	0.088ms	
300px-Colored...	upload.wikime...	Image	GET	HTTPS	200	Yes (Mem...	32.23 KB	0 B	745.2ms	0.161ms	0.057ms	
290px-Linear_r...	upload.wikime...	Image	GET	HTTPS	200	Yes (Mem...	7.42 KB	0 B	746.5ms	0.123ms	0.068ms	
220px-SimpleR...	upload.wikime...	Image	GET	HTTPS	200	Yes (Mem...	7.62 KB	0 B	747.1ms	0.130ms	0.051ms	

Localhost



More info here.

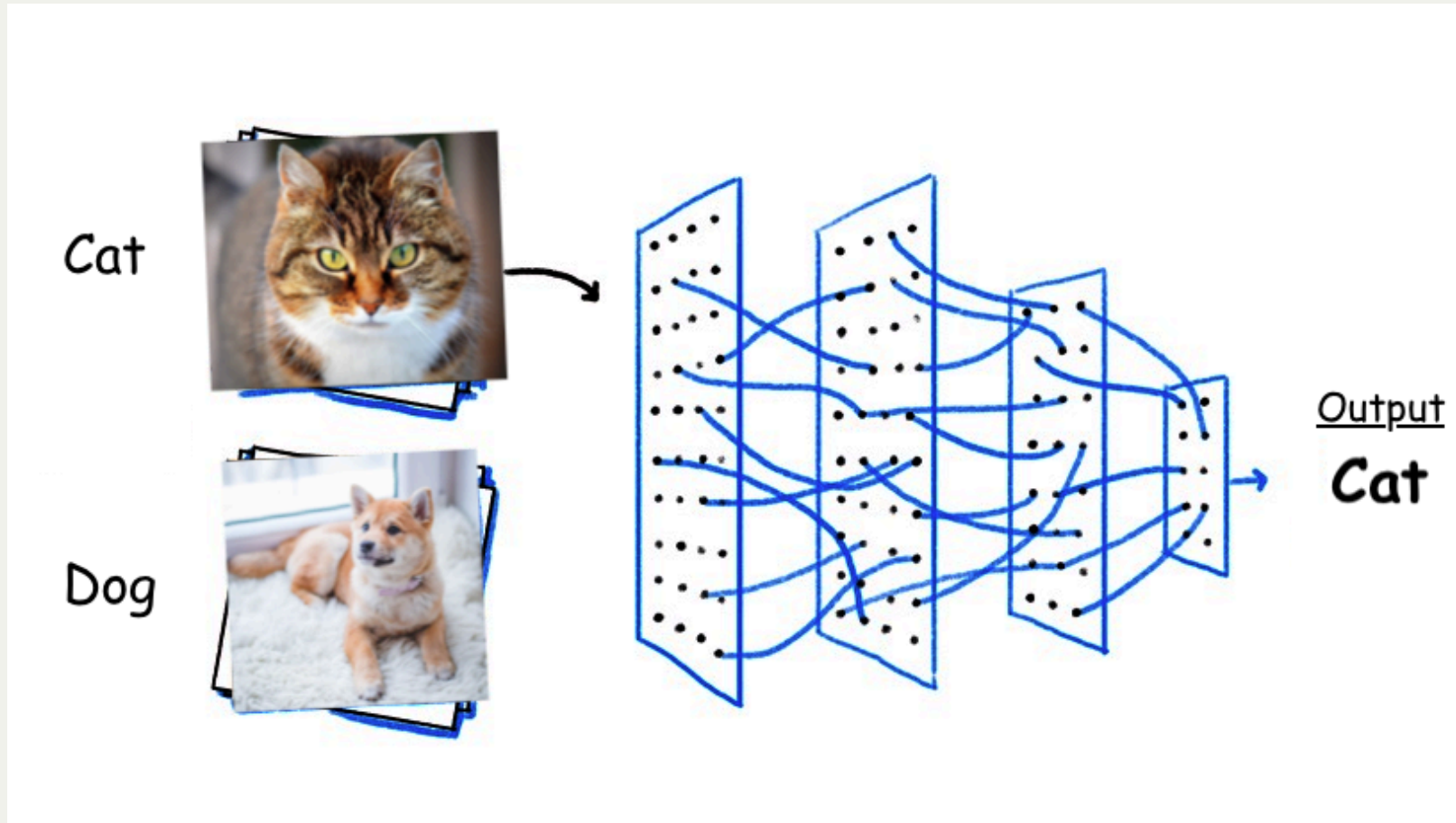
Deployment



deploy resources = make them ready to be used

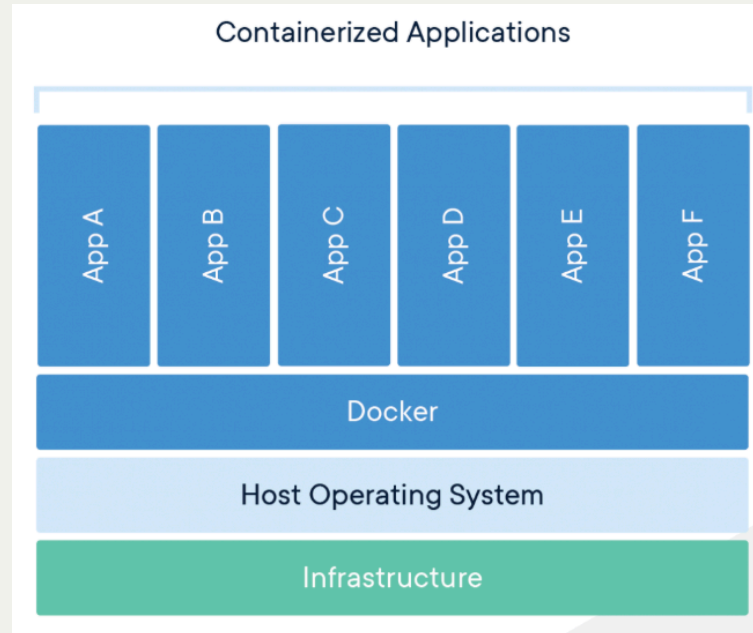
We will not deploy our application for this tutorial.

Image classification



Want to know more? Check out this [tutorial on image classification with PyTorch](#).

Containers



Some information about containers.

Not only Docker...

Part 1: Define the service

First, we have to define what we want to build.

Our **requirements** are:

- **a web app that runs a simple ML algorithm for image classification**
- inside a **container** - don't worry about it now
- **time constraint** (always take into account)

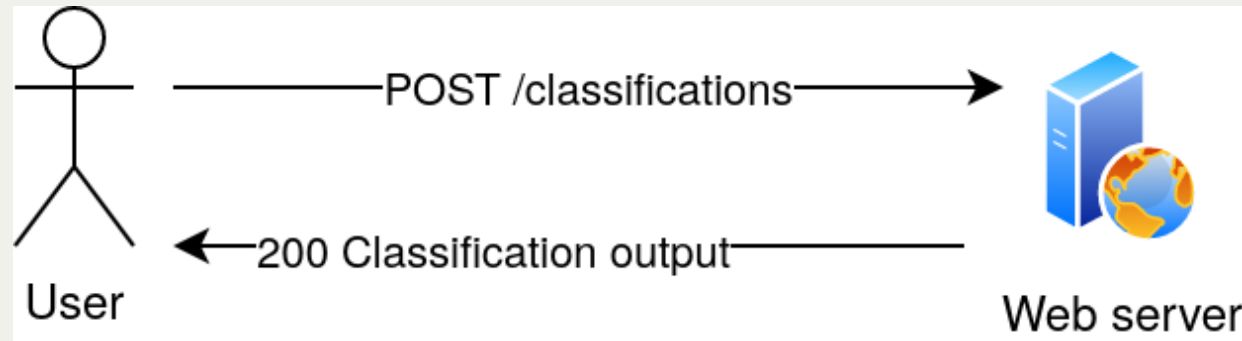
Before start writing any code ...

This is an important part of our development process. If we rush into writing the code, the risk is to waste time.

Better stop and take a moment to think what is the structure of our application.

Use cases

The user should be able to classify an image.



What can the user change? What is fixed?

We decide that the user can only choose a specific model and a specific image from a set of models and a set of images.

Modern ML systems are very fast but...

What if classifying the image takes longer?

What could go **wrong** in our demo?

The user expects a quick response

It's not necessary to provide the result already, but we need to tell the user we heard the request.

Otherwise, the user might get annoyed and send multiple requests.

We want to avoid that.

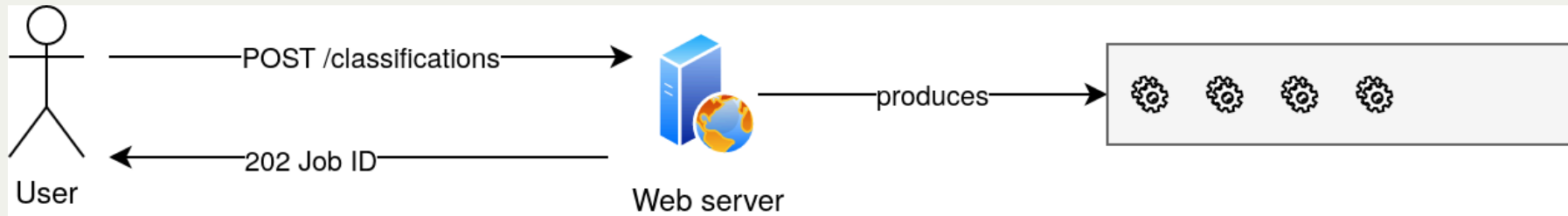
What is the solution?

The webserver enqueues the job and returns to the user a "**ticket**" for getting the results. The "ticket" will be the ID of the job.



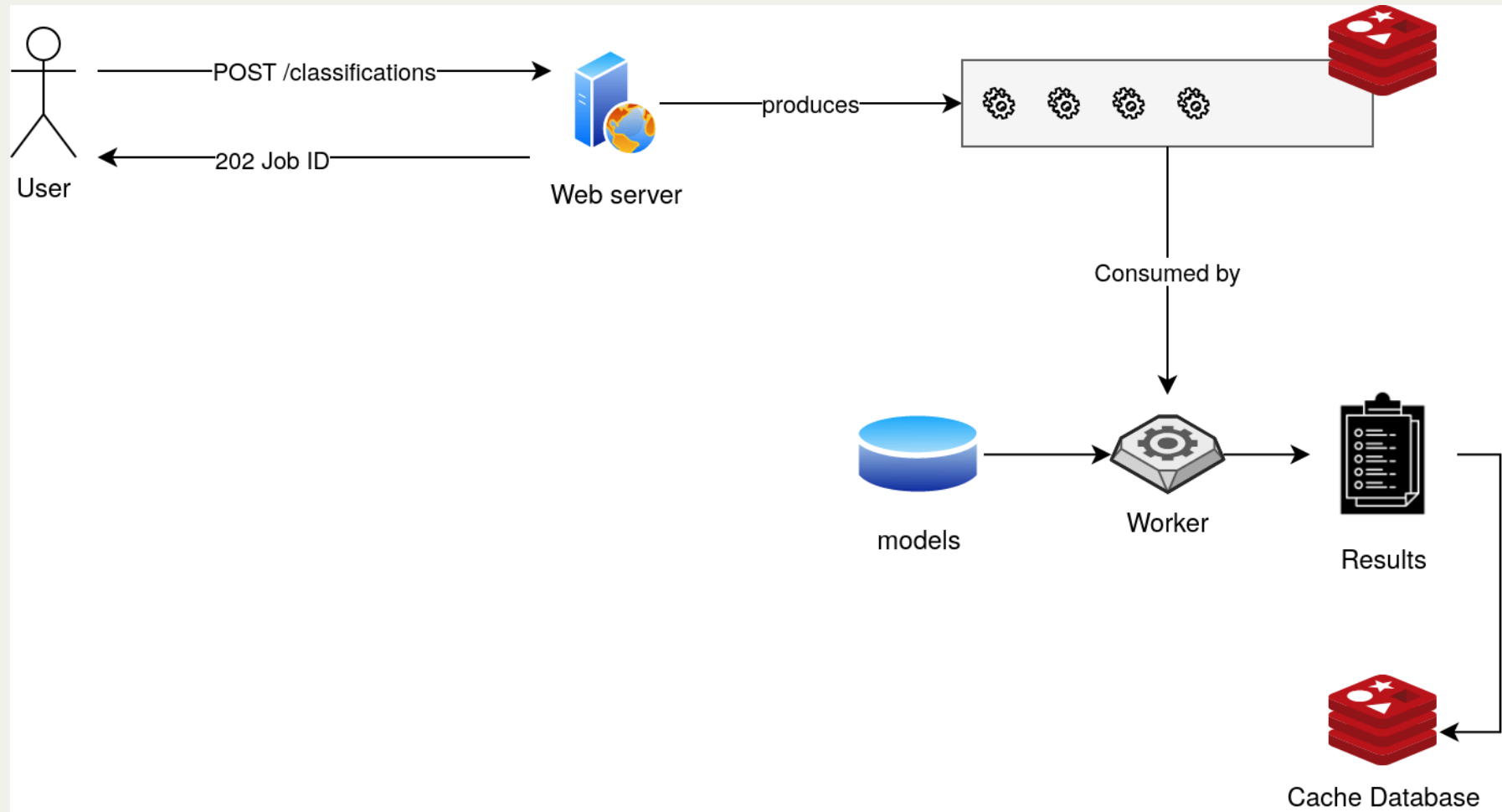
We will implement **asynchronous** jobs.

We create a **queue**, save the request, and store the results when they are ready. We will use a simple database for handling the queue.

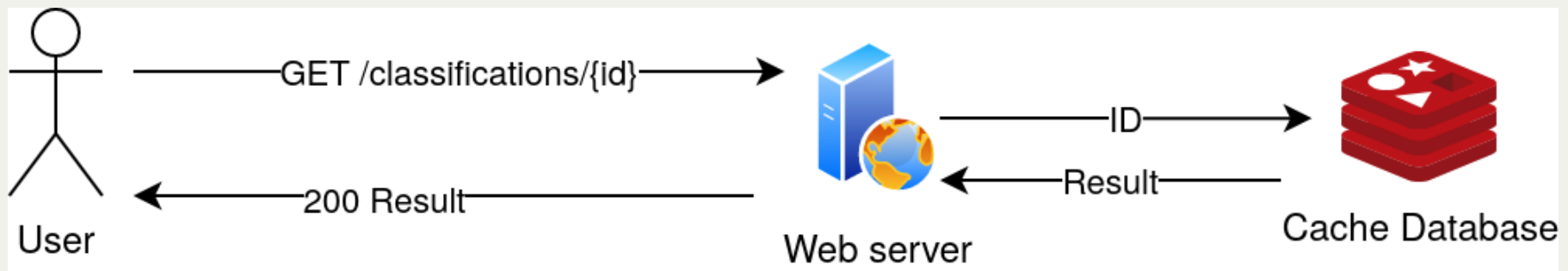


The **worker**, another service of our webserver, takes the enqueued jobs with a **FIFO** (First-In-First-Out) schedule, and processes the requests.

Once completed, each job result is stored in the database, with the job ID as Key for accessing the newly-produced data.

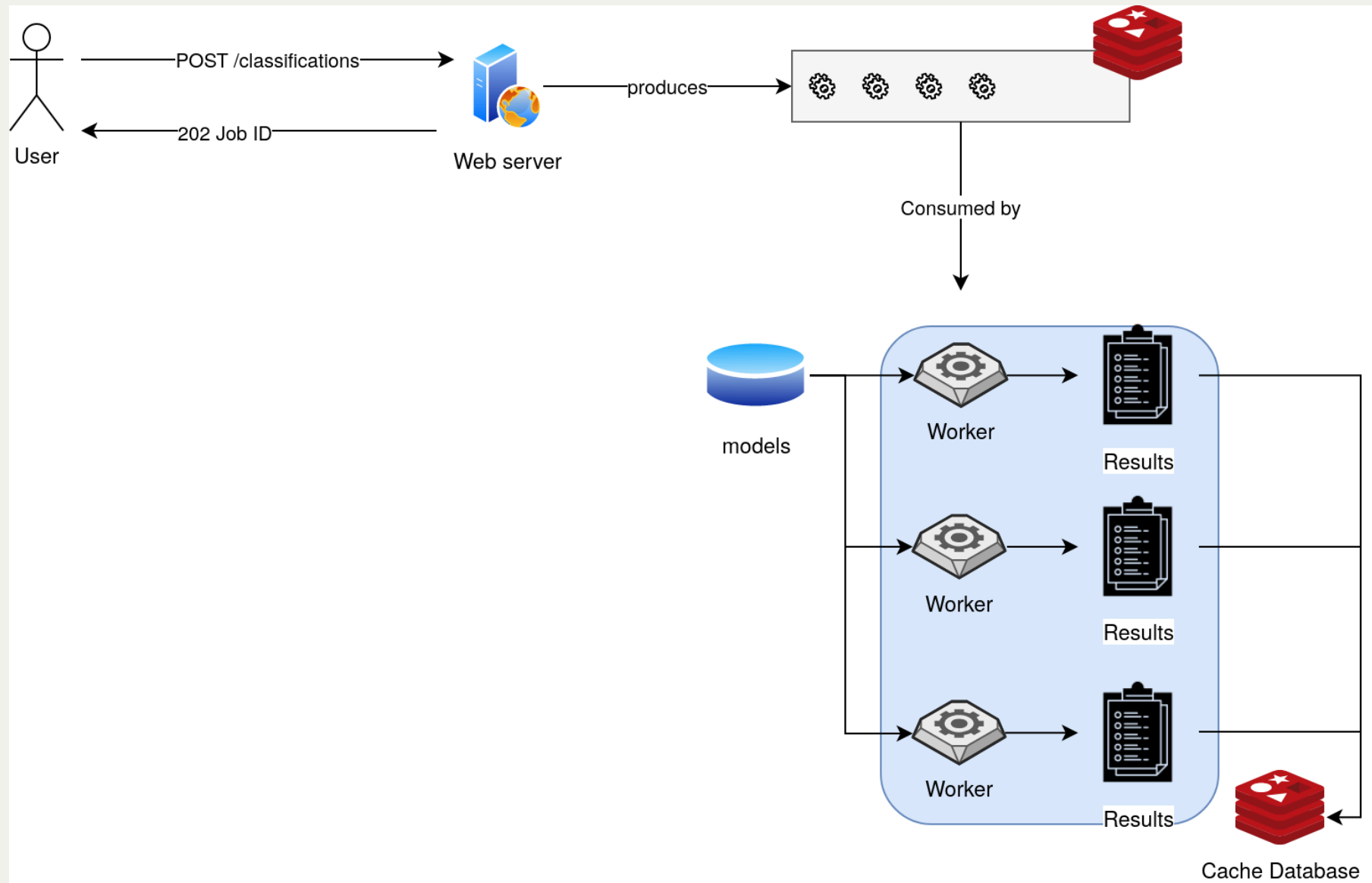


After some (short) time, the user should be able to send a request to the server, providing the job id, and getting the results as a response.



What are the advantages of enforcing **modularity**?

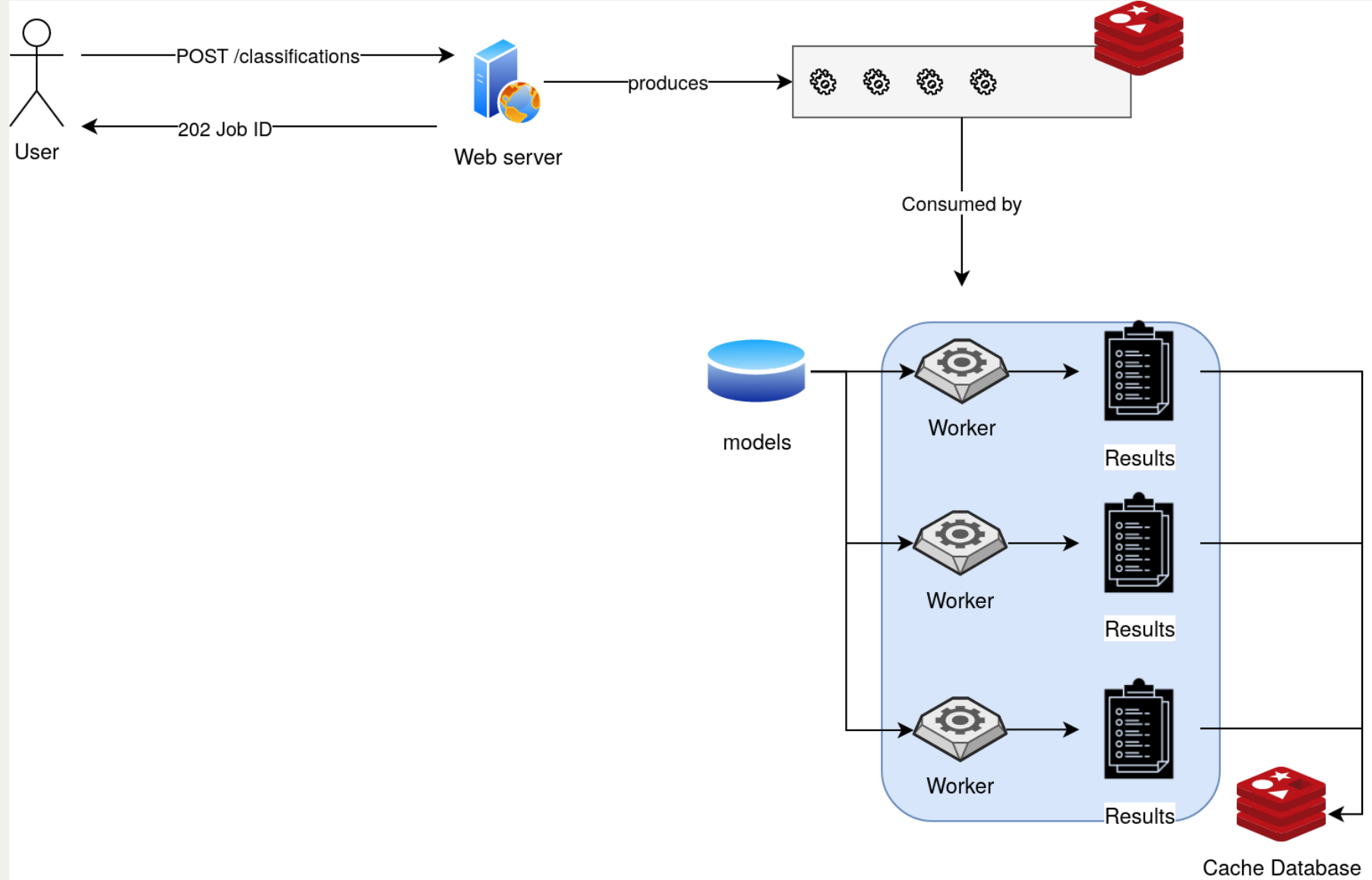
- failures are isolated to single components
- scaling is easier



We can still make another improvement: pre-downloading the images and the models.

What pieces of our architecture should be able to access them?

(the architecture)



Notice something...

We haven't even named a single software until now...

For what is worth, our application might not even be written in Python!

Now we can introduce tools can help us design and maintain our code.

...still no code yet!

Tools for developers

- GitHub: service that hosts the versioned source code of our application.
- Swagger: tool for designing and **documenting** APIs, using the Open API specifications.

We are not going to write the API definition in swagger, but this is how they look like:

```
paths:
  /info:
    get:
      summary: Information about available models and images.
      description: Returns the list of pretrained models available and the metadata of the gallery images.
      responses:
        '200':
          description: Available models and images.
          content:
            application/json:
              schema:
                type: object
                properties:
                  models:
                    type: array
                    items:
                      type: string
                      example: "vgg16"
                  images:
                    type: array
                    items:
                      type: string
                      example: "n01531178_goldfinch.JPEG"
```

This is written in YAML. We will see another one in this lesson.

And here we can find the APIs we have to create, rendered by Swagger.

Now we can start creating our building blocks

Building blocks

- a web server
- a queue
- some worker

To achieve scalability:

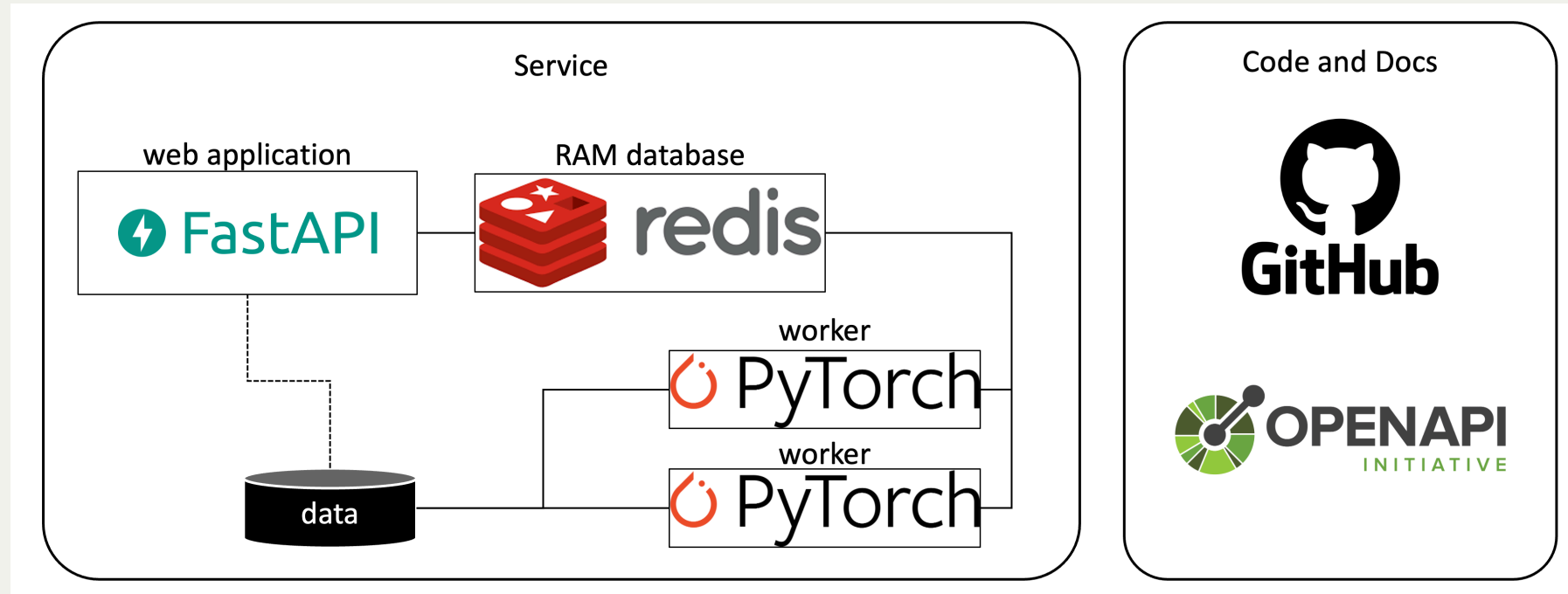
- a "box"
- some storage

Building blocks (with a name)

- FastAPI (a web server)
- Redis (a queue)
- Python + PyTorch (some worker)

To achieve isolation:

- Docker (a "box")
- Docker volumes (some storage)



This seems a very complicated architecture, but we are lucky! Docker has the perfect tool for this!

Docker-compose interconnects several containers through APIs.

Now that we have a rough idea of what are the steps, we can start writing some code!

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=3.10  
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:

```
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:

```
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:

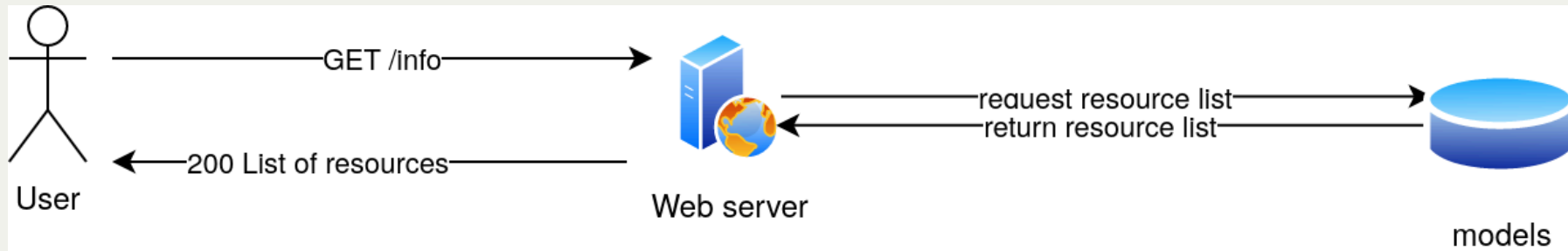
<http://127.0.0.1:8000/docs>

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.



```
from typing import Dict, List
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, right¹? 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.


```
from fastapi import FastAPI, Request
from fastapi.templating import Jinja2Templates
from fastapi.responses import HTMLResponse
from fastapi.staticfiles import StaticFiles

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 inputs².

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

```
from typing import List
from fastapi import Request

class ClassificationForm:

    def __init__(self, request: Request) -> None:
        self.request: Request = request
        self.errors: List = []
        self.image_id: str = ""
        self.model_id: str = ""

    async def load_data(self):
        form = await self.request.form()
        self.image_id = form.get("image_id")
        self.model_id = form.get("model_id")

    ...
```

```
class ClassificationForm:

    ...

    def is_valid(self):
        if not self.image_id or not isinstance(self.image_id, str):
            self.errors.append("A valid image id is required")
        if not self.model_id or not isinstance(self.model_id, str):
            self.errors.append("A valid model id is required")
        if not self.errors:
            return True
        return False
```


We can then create the classification request

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.

```
@app.get("/classifications")
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']`.

```
@app.post("/classifications")
async def request_classification(request: Request):
    form = ClassificationForm(request)
    await form.load_data()
    classification_scores = classify_image(
        model_id=form.model_id,
        img_id=form.image_id)
    if form.is_valid():
        data = form.__dict__
        data['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?

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

```
1 from redis import Redis
2 from rq import Connection, Queue
3 from rq.job import Job
4
5 @app.post("/classifications")
6 async def request_classification(request: Request):
7     form = ClassificationForm(request)
8     await form.load_data()
9     image_id = form.image_id
10    model_id = form.model_id
11    redis_conn = Redis(config.REDIS_HOST, config.REDIS_PORT)
12    q = Queue(name=Configuration.QUEUE, connection=redis_conn)
13    task = q.enqueue(classify_image, model_id=model_id, img_id=image_id)
14    return templates.TemplateResponse(
15        request, "classification_output_queue.html",
16        {"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}`.

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

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

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):

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...

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 documentation³.

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

Containers

For creating a container with Docker, we use a specific file called `Dockerfile`. This file is automatically understood by Docker and it has a specific format.

We are not going to write one from scratch, but we can inspect the one that builds our application.

```
FROM python:3.10
```

```
# We copy just the requirements.txt first to leverage  
# Docker cache
```

```
COPY ./requirements.txt ./requirements.txt
```

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

```
ADD . ./
```

```
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "80"]
```

The important part here is to remember that:

- we are starting from an image that already contains Python and some other useful tools, *e.g.*, `pip`.
- we are leveraging Docker cache.

Docker cache

Docker builds **intermediate containers** for every line we have in this Dockerfile.

If we change the content of one of the lines, Docker uses the cached version of everything before the changed line and rebuilds what comes after the line.

Building the container

Now let's open a terminal in the root directory, and run:

```
docker build . -t classification-ws
```

We are telling Docker to build the current directory, and to **tag** the image we just created with the name `classification-ws`.

Docker will automatically search for a file `Dockerfile` in the specified directory.

If you haven't done it yet, remember to stop the webserver that we were using until now.

This is because we will run the same service through the docker container now!

Then, we can run the container with the command:

```
docker run -p 80:80 classification-ws
```

Note that we are specifying here a **socket**. This is a mapping of a port inside the container with a port in our computer.

So, inside our container we will run the server on port 80, which will be linked with the port 80 of our localhost.

We can see that the container is running the server in our localhost port 80⁴.

Don't run classifications there yet...

⁴ note that port 80 is the default for the web browser

What happens when we click on "submit"?

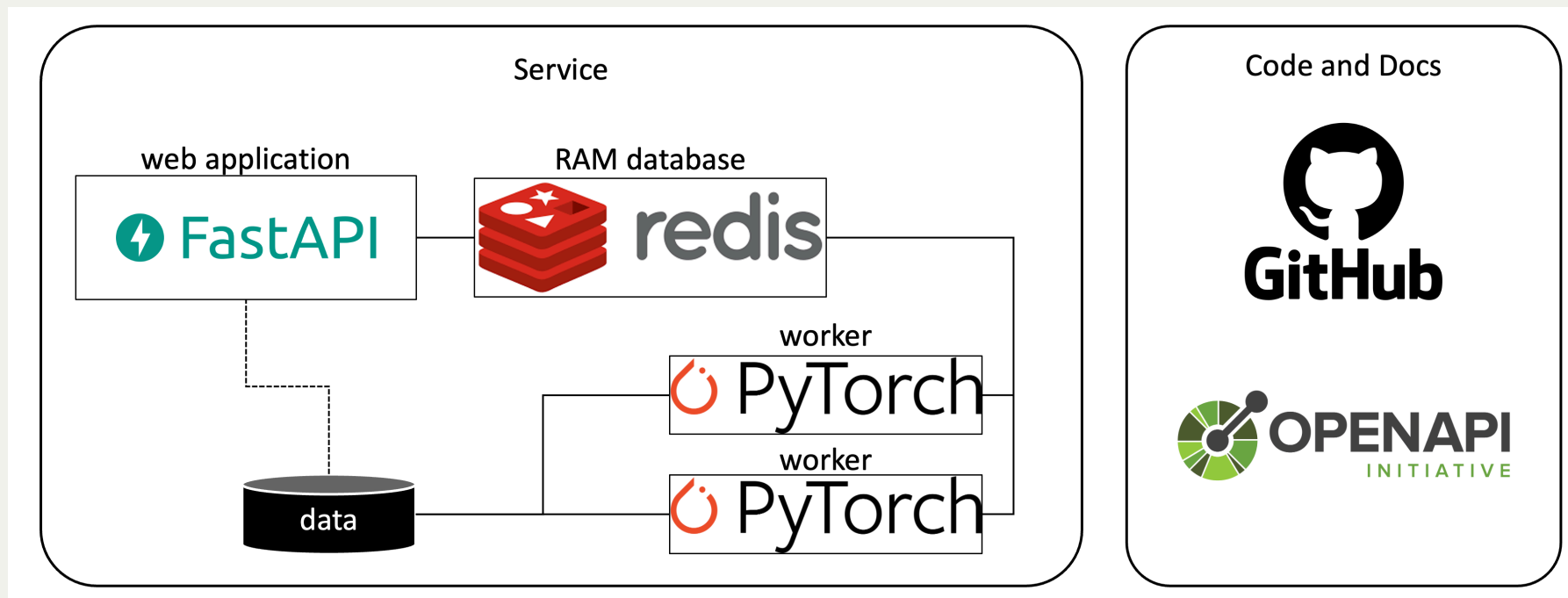
We should improve our docker service a little bit, right now it is missing the redis service, the volume, and the worker.

If we ask for a classification job now, we will be stuck with an error...

Stop the container with Ctrl+C.

Docker compose

Remember the architecture? See how many containers are there:



See how many containers are there now...

- one for the web application
- one for each worker
- one for the redis database

If we want to define more than one container and link them together, we should use a docker compose file.

We have one already in our root-directory. Let's inspect that.

We have three blocks:

- web
- redis
- worker

```
web:
  build: .
  command: uvicorn main:app --host 0.0.0.0 --port 80
  ports:
    - "80:80"
  links:
    - redisdb
  environment:
    - REDIS_HOST=redisdb
    - REDIS_PORT=6378
  volumes:
    - ~/.cache/torch:/root/.cache/torch
```

The `web` container is running the webserver on port 80.

It has a `build .` command that is similar to what we just did with the standalone container, and some other interesting keywords.

links defines the connection of this container with others defined in the same dockerfile.

We are connecting this container with the one running the database.

environment defines environment variables, that we can use for storing dynamic values like the redis port and the hostname.

Why is the hostname `redisdb`?

By default, docker links define an entry in the hosts file of our containers that points to the linked containers.

So if we connect to `resdisdb` from inside of our `web` container, we will see the localhost of the `redisdb` container.

volumes is another interesting trick. We are mounting a directory from our **filesystem** into the container's filesystem.

This means that the files located here persist even when the container is stopped.

We are using this trick to avoid downloading models every time we run the container.

Now check the remaining parts of the docker-compose file.
You should now be able to understand them.

```
redisdb:  
  image: "redis"  
  command: --port 6378  
  ports:  
    - "6378:6378"
```

```
worker:
  build: .
  command: python worker.py
  links:
    - redisdb
  environment:
    - REDIS_HOST=redisdb
    - REDIS_PORT=6378
  volumes:
    - ~/.cache/torch:/root/.cache/torch
```

And finally, let the magic happen! We can create our architecture with a single line:

```
docker-compose build && docker-compose up
```

What is the beauty of our docker compose? First, we can download the whole repository and install it in the client's computer without sweating too much...

Moreover, we can also easily scale our service, for example by running 2 workers instead of one!

```
docker-compose up --scale worker=2
```

There are other improvements that can be easily implemented with this architecture.
Can you figure out them?

- scale web container and add load balancer
- caching machine learning results

Summary

- design phase of a project
- frontend-backend
- APIs
- long running jobs and queues
- containers