Image Classifier App - Part 1

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 lectures will cover:

- basic concepts of REST APIs and web applications
- software design
- inspecting code written by others
- implementing a few basic APIs
- containerization
- 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.*

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.*

Spoiler: the ML team also developed an LLM-based chatbot...

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**.

Input
Black Box
Output

We are not going to start from scratch.

The team has a repository that contains already some code.

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, REST APIs, image classification

Applications Landscape

Modern digital ecosystems include:

- **Consumer apps**: AI, e-commerce, social networks, messaging, streaming, travel booking.
- **Enterprise apps**: customer relationship, resource planning, recruitment, analytics dashboards.
- **Device/IoT**: smart sensors and devices, wearables, industrial sensors, home assistants.
- **Automation & Integrations**: payment processing, shipping, notifications, identity.

Common Theme: These systems **exchange structured data** across organizational and network boundaries.

Applications Landscape

Modern digital ecosystems include:

- **Consumer apps**: AI, e-commerce, social networks, messaging, streaming, travel booking.
- **Enterprise apps**: customer relationship, resource planning, recruitment, analytics dashboards.
- **Device/IoT**: smart sensors and devices, wearables, industrial sensors, home assistants.
- **Automation & Integrations**: payment processing, shipping, notifications, identity.

Common Theme: These systems **exchange structured data** across organizational and network boundaries.

Frequent Patterns

- **Request/response**: a system asks, another replies.
- **Event-driven**: systems emit events; others subscribe.
- **Batch transfers**: periodic file or dataset exchange.
- **Webhooks/callbacks**: notify peers when something changes.

Examples of distributed architectures over the web

Most web applications:

- Render **UI** on web or mobile.
- **Call** remote servers to fetch or mutate data.
- **Compose** results from multiple internal and external services.
- **Persist** to databases, message queues, and object stores.

Examples of distributed architectures over the web

Most web applications:

- Render **UI** on web or mobile.
- **Call** remote servers to fetch or mutate data.
- **Compose** results from multiple internal and external services.
- **Persist** to databases, message queues, and object stores.

Cloud applications run on infrastructure managed by a provider, offering on-demand compute, storage, networking, and other functionalities with elasticity and managed services.

- **Models**: X-as-a-service (where X can be: Infrastructure, Platform, Software, etc.)

Examples of distributed architectures over the web

Most web applications:

- Render **UI** on web or mobile.
- **Call** remote servers to fetch or mutate data.
- **Compose** results from multiple internal and external services.
- **Persist** to databases, message queues, and object stores.

Cloud applications run on infrastructure managed by a provider, offering on-demand compute, storage, networking, and other functionalities with elasticity and managed services.

- **Models**: X-as-a-service (where X can be: Infrastructure, Platform, Software, etc.)

Microservices architectures decompose a system into **small, autonomous services** that communicate over the network.

The Client?

A **client** is software that **initiates** communication to request data or actions. Examples:

- Browsers (Chrome, Edge, Firefox) rendering web apps.
- Mobile apps requesting user profiles, feeds, payments.
- Server-side jobs integrating with external systems.

Responsibilities: construct requests, handle responses, manage retries, respect timeouts, secure secrets.

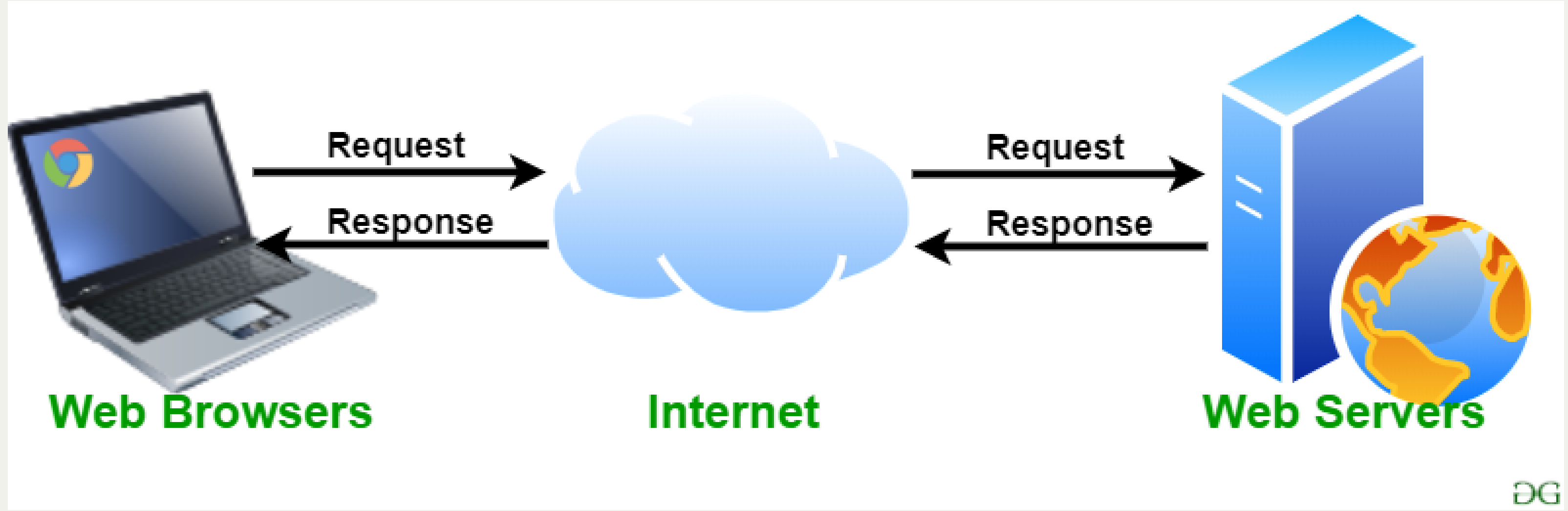
The Server?

A **server** is software that **listens** for requests and responds with data, actions, or errors. Examples:

- Web application backends exposing business capabilities.
- Media servers serving images, video segments.
- Identity servers handling sign-in, tokens.

Responsibilities: validate input, enforce authorization, execute business logic, persist data, produce responses.

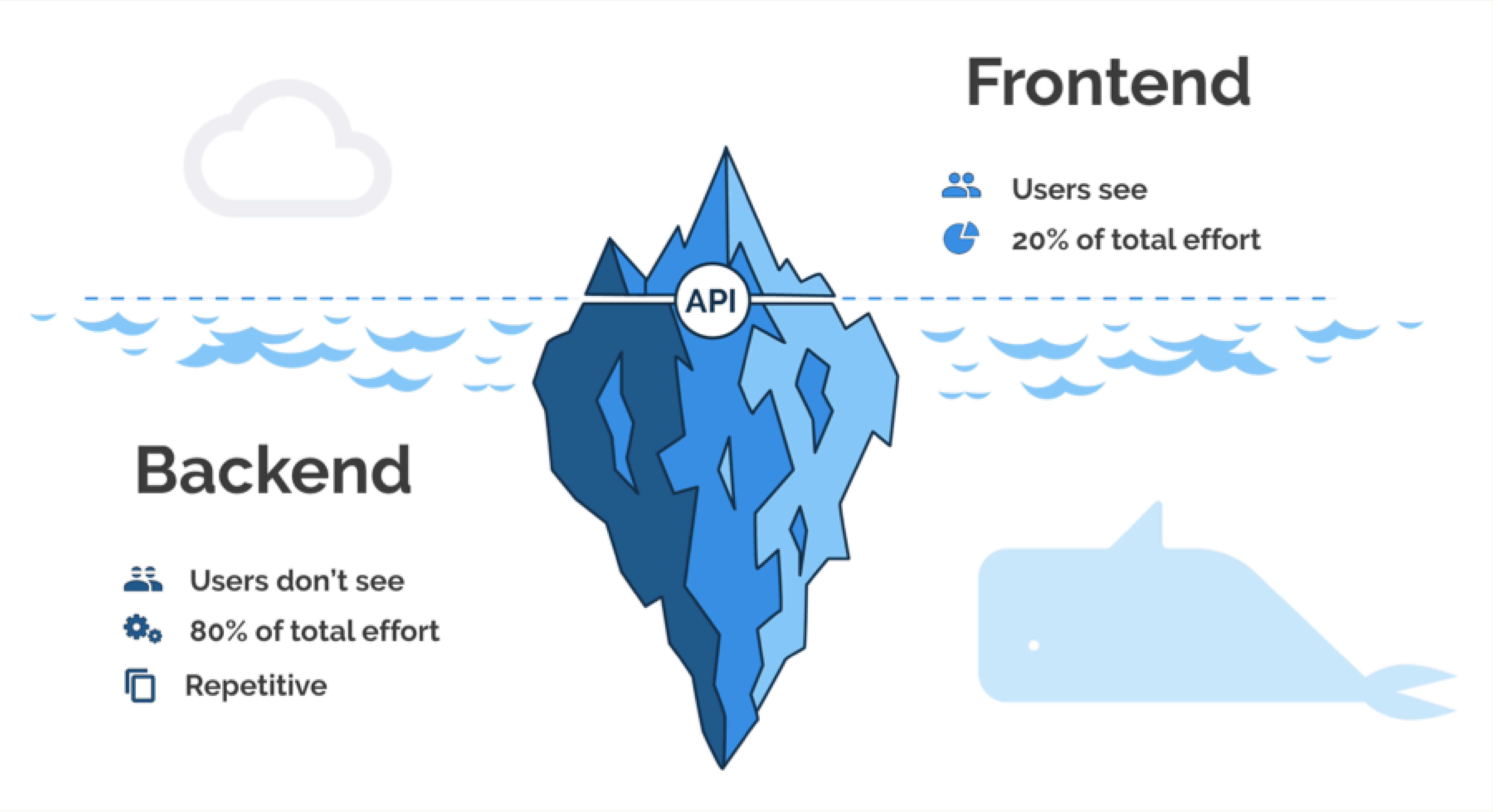
Web server for the user



More info here.

Example of webserver

Web server for the developer



More info here.

Client-server interaction

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.

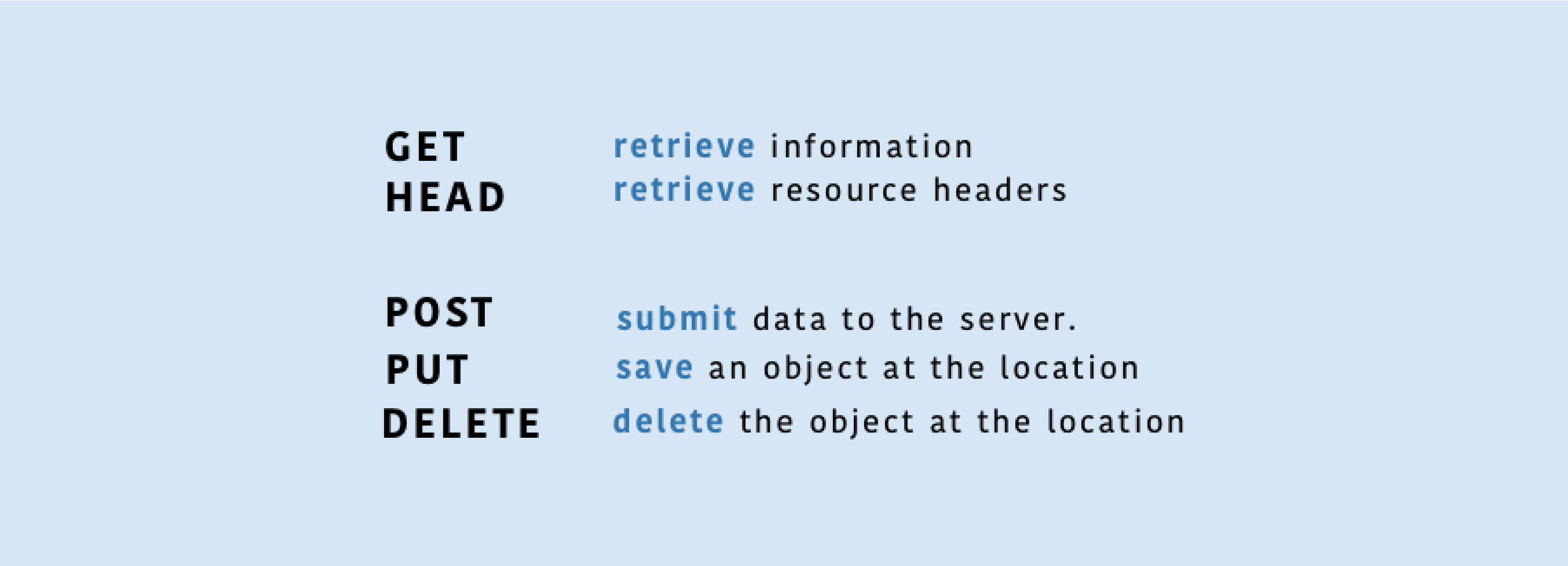
Key properties: clear roles, decoupling across a network, independent scaling.

The language of web servers

The dominant communication channel among these applications is **HTTP**:

- Ubiquitous support in browsers, mobiles, servers, etc.
- Works seamlessly over the internet and internal networks.
- Originally created for hypertext documents on the early web.
- Evolved from simple text retrieval to a flexible application protocol for data and 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

- Not only web pages: programmatic data exchange between services.
- Rich semantics: **methods**, **status codes**, **headers**, **representation formats**.

HTTP is Stateless

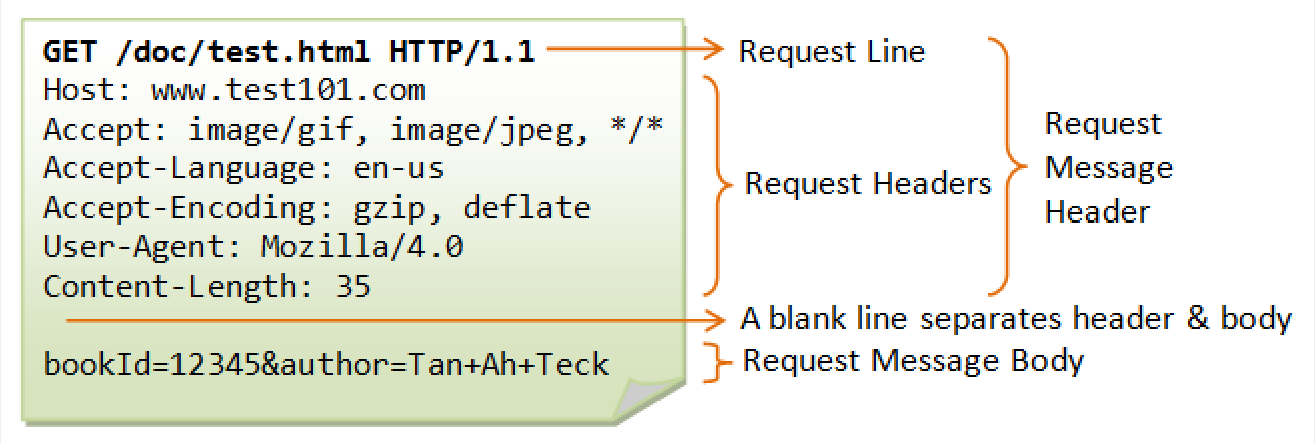
It means that:

- Each HTTP request is **independent**.
- The server does not remember previous requests.
- No built-in session or memory.

Implications:

- Every request must include all necessary data (e.g., authentication).
- State is managed via **cookies, sessions, tokens**.

HTTP request

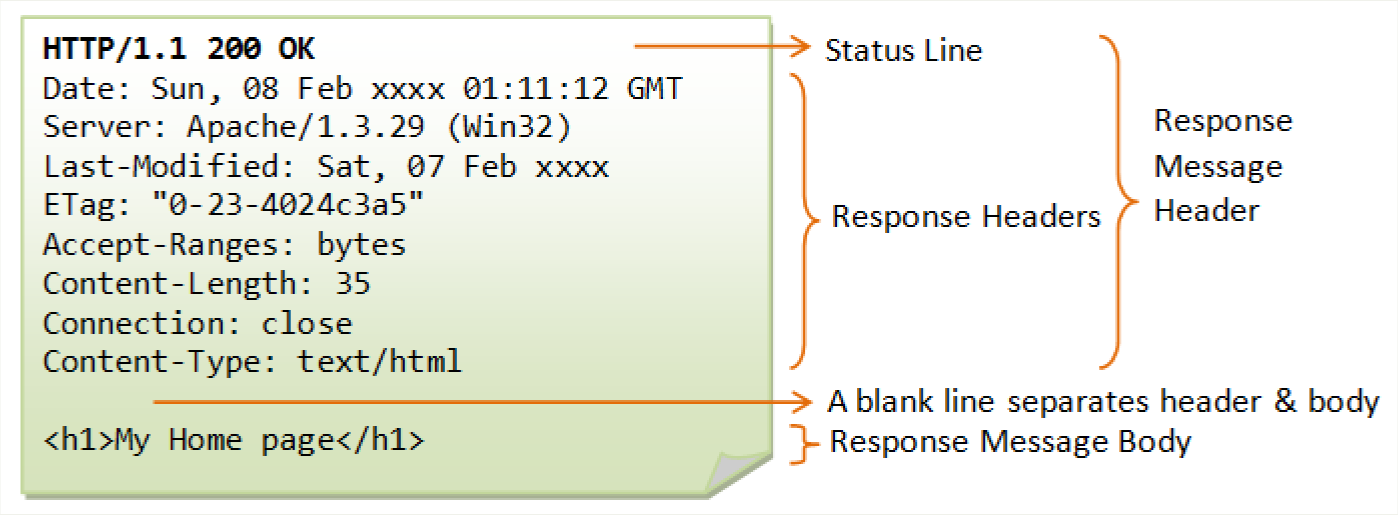


```
GET /doc/test.html HTTP/1.1
Host: www.test101.com
Accept: image/gif, image/jpeg, */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0
Content-Length: 35

bookId=12345&author=Tan+Ah+Teck
```

More info on HTTP Protocol.

HTTP response



```
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>
```

Status Line

Response Headers

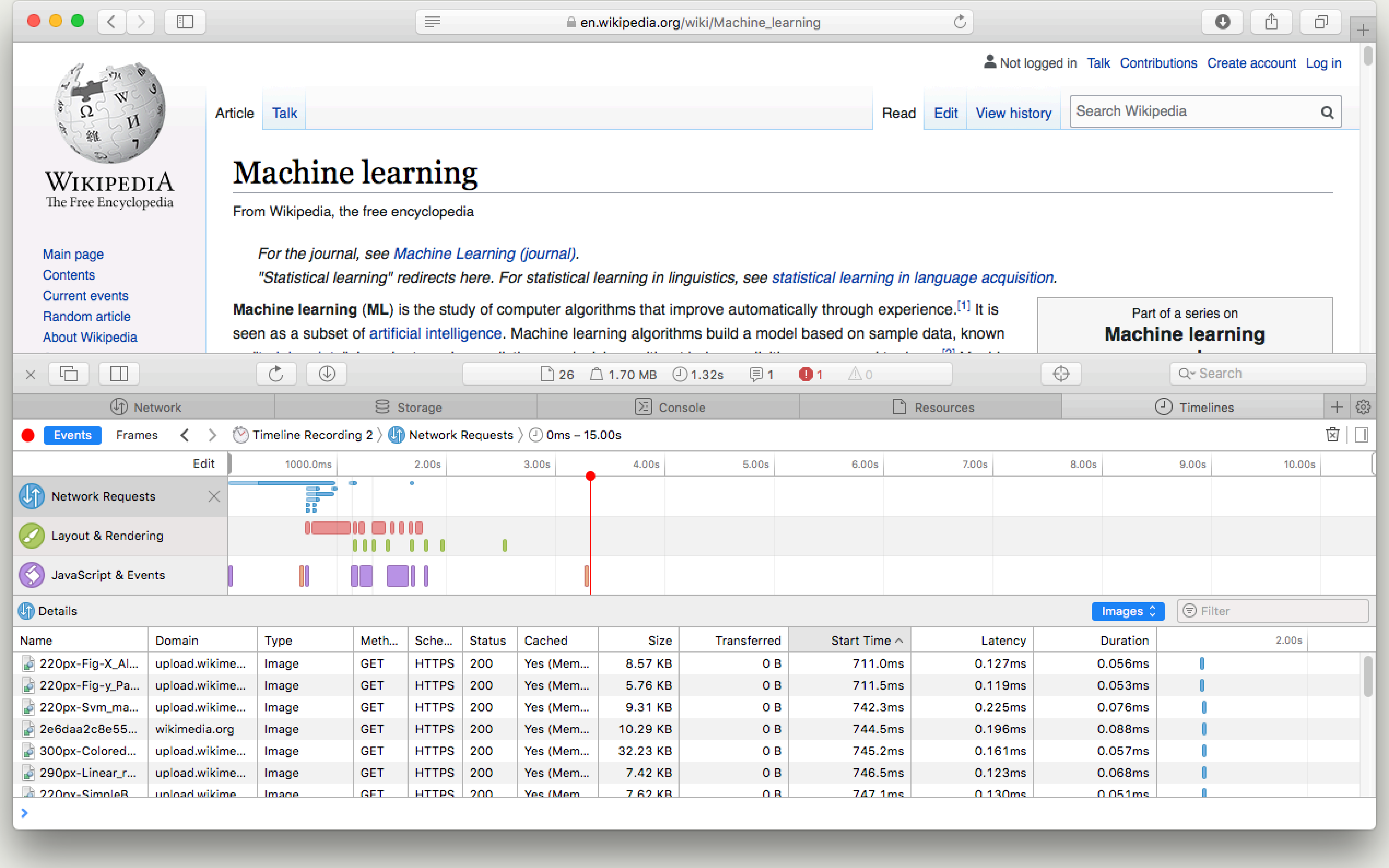
Response Message Header

A blank line separates header & body

Response Message Body

Status Codes - Categories

- **1xx Informational**: processing continues.
- **2xx Success**: 200 OK, 201 Created, 204 No Content.
- **3xx Redirection**: 301 Moved Permanently, 304 Not Modified.
- **4xx Client Errors**: 400 Bad Request, 401 Unauthorized, 403 Forbidden, 404 Not Found.
- **5xx Server Errors**: 500 Internal Server Error, 502 Bad Gateway, 503 Service Unavailable.



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

Name	Domain	Type	Meth...	Sche...	Status	Cached	Size	Transferred	Start Time	Latency	Duration
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-SimpleB...	upload.wikime...	Image	GET	HTTPS	200	Yes (Mem...	7.62 KB	0 B	747.1ms	0.130ms	0.051ms

API

What is an API?

Open video

More info about APIs.

REST APIs

Representational State Transfer (REST) is an architectural style for designing APIs that leverages:

- **Resources** identified by **URIs**.
- **Representations** (e.g., JSON) transferred via stateless interactions.
- Use of **standard HTTP semantics** for actions.
- **Hypermedia** as a way to discover next actions.

Resources and URIs

- Model **nouns** (things) not verbs: `/users`, `/orders`, `/orders/{id}`.
- **Hierarchy** and relations: `/customers/{id}/orders`.
- **Collection vs item**: collections respond to listing and creation; items to read/update/delete.

Example

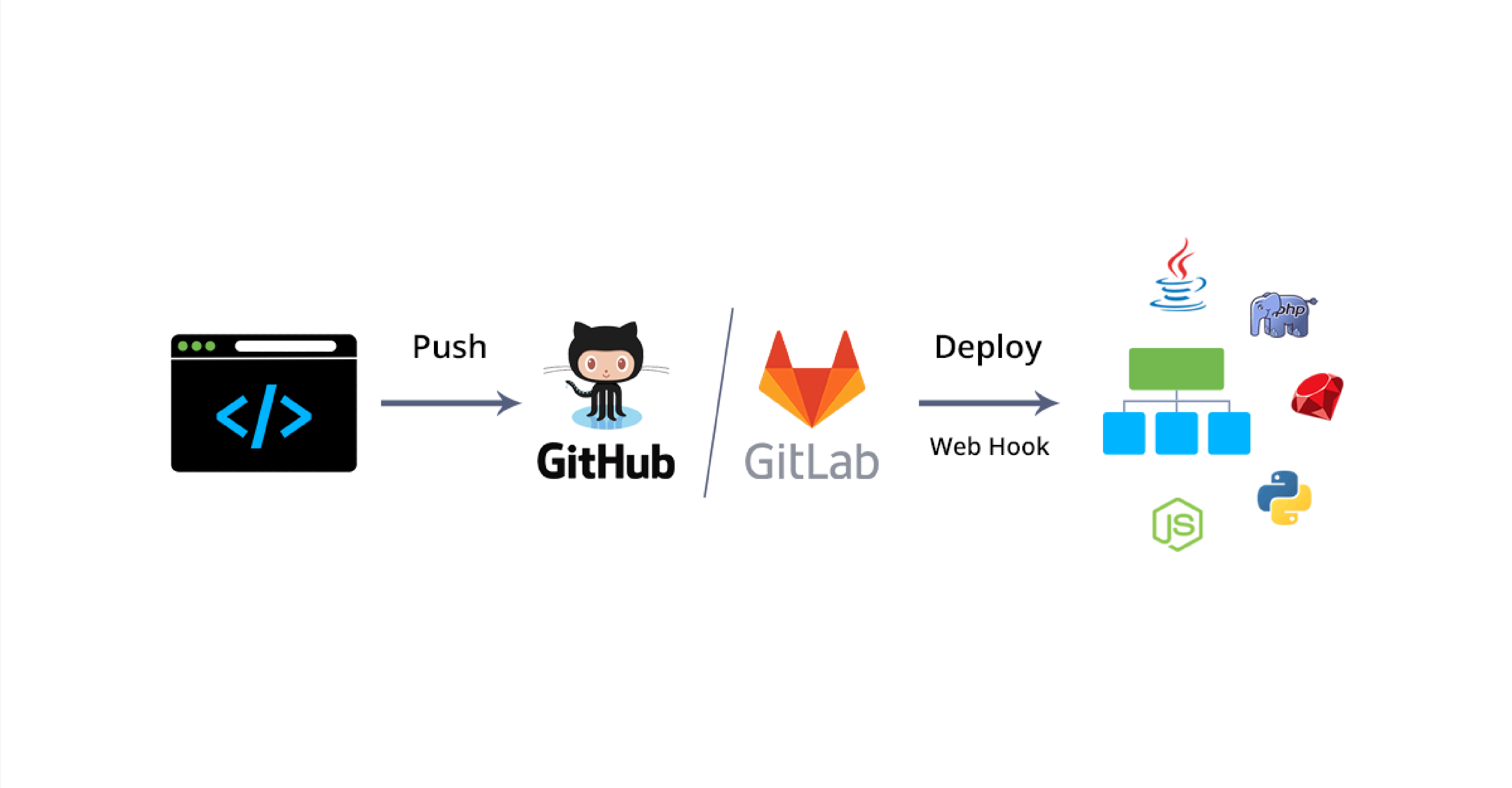
Methods Mapped to Resource Semantics:

- **GET /orders**: list orders (filter/sort/paginate via query params).
- **POST /orders**: create an order (returns `201 Created` + `Location`).
- **GET /orders/{id}**: retrieve one.
- **PUT /orders/{id}**: replace entire order.
- **PATCH /orders/{id}**: partial update.
- **DELETE /orders/{id}**: remove.

Validation and Error Handling

- **Consistent error schema**: code, message, details, correlation ID.
- **Use appropriate status codes** and avoid overloading 200.
- **Input validation**: types, ranges, formats; return actionable messages.

Deployment



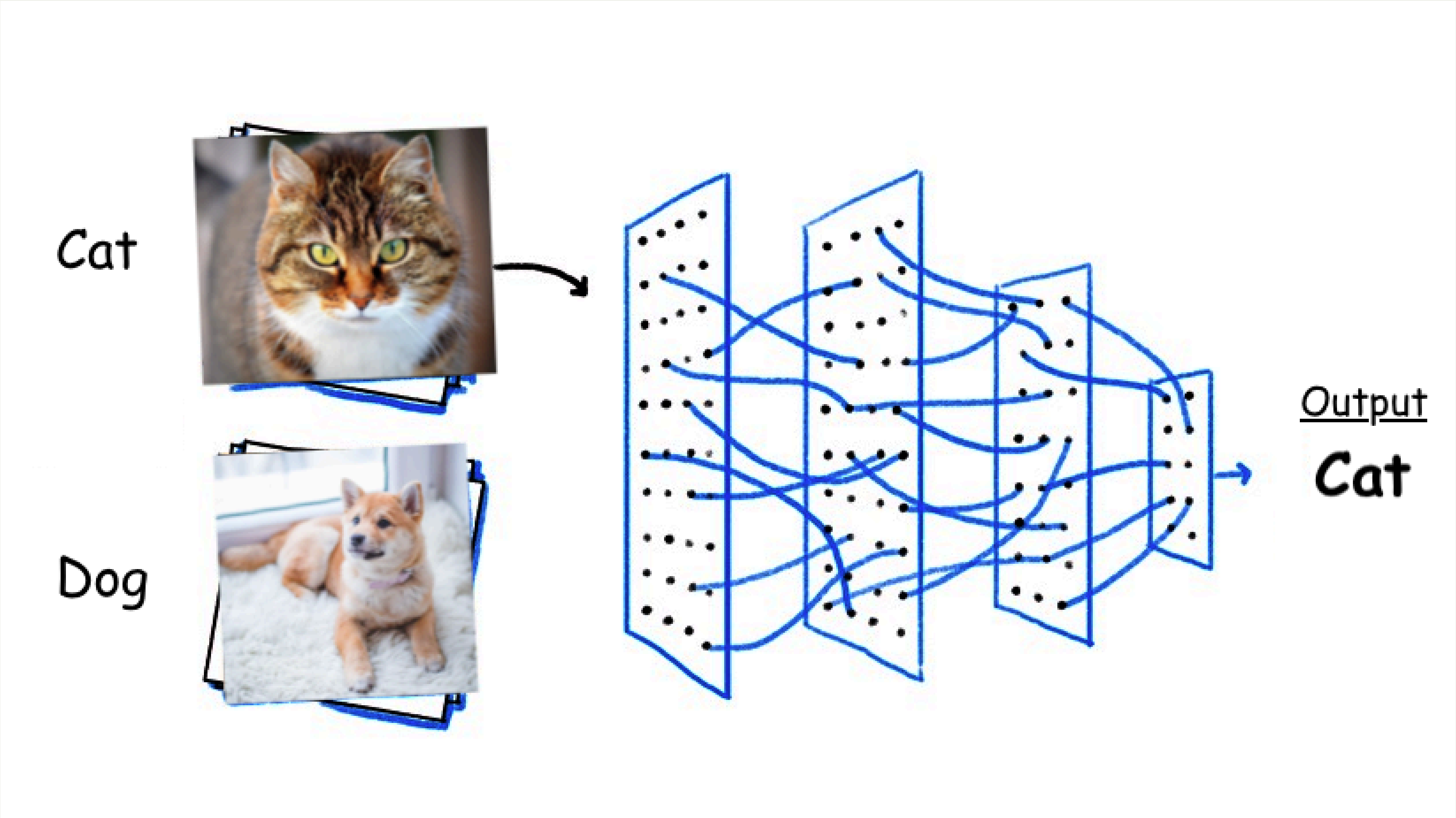
deploy resources = make them ready to be used

We will not deploy our application for this tutorial.

Localhost

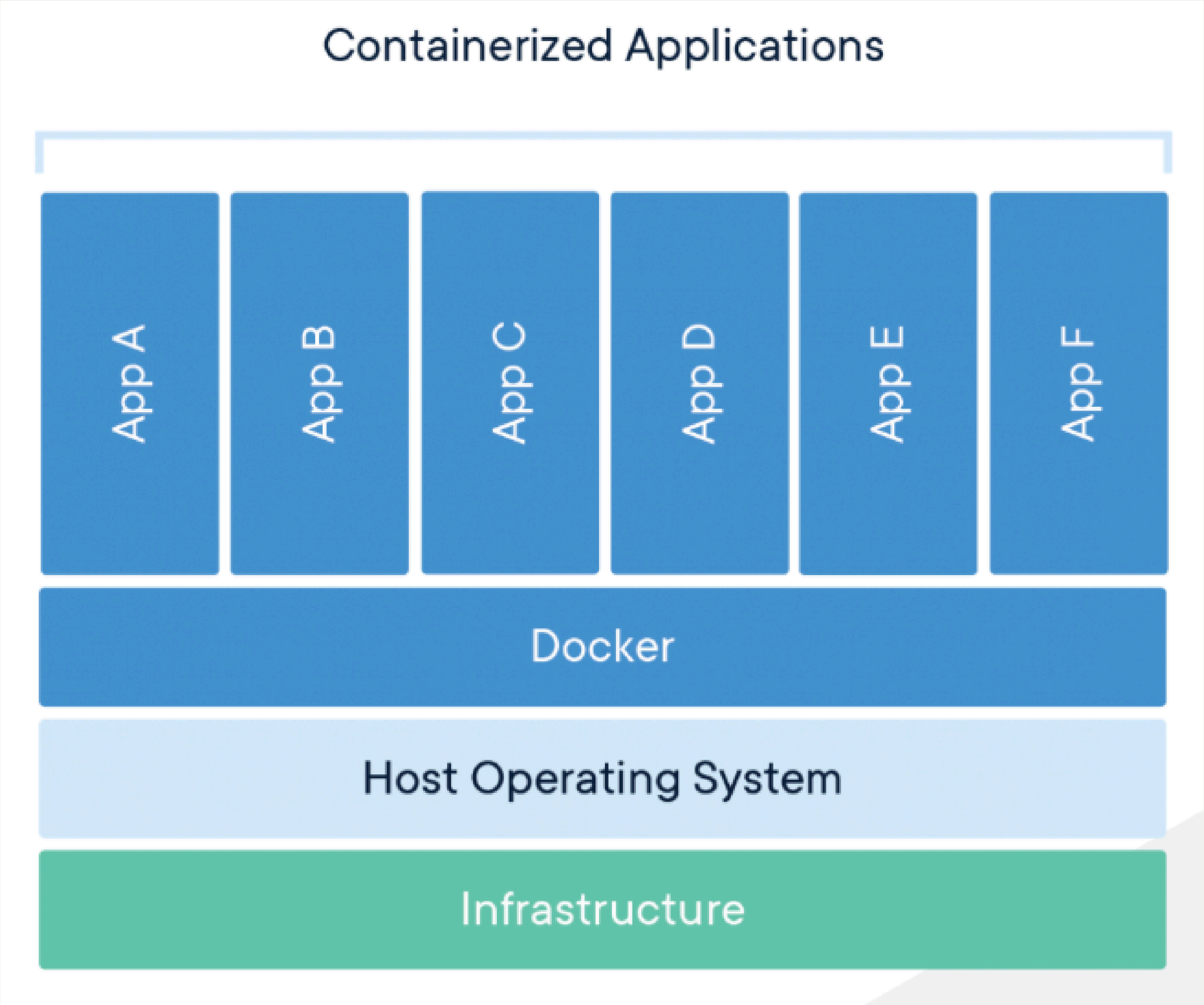
More info here.

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)

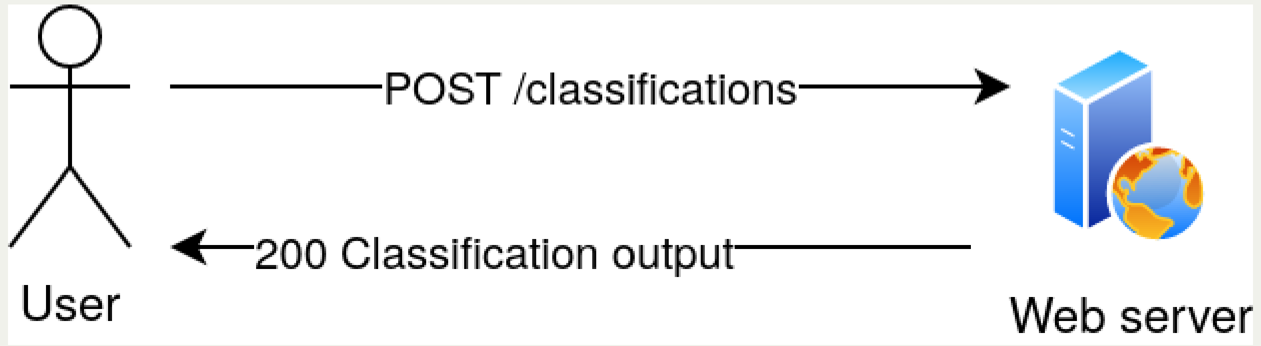
Before start writing any code ...

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

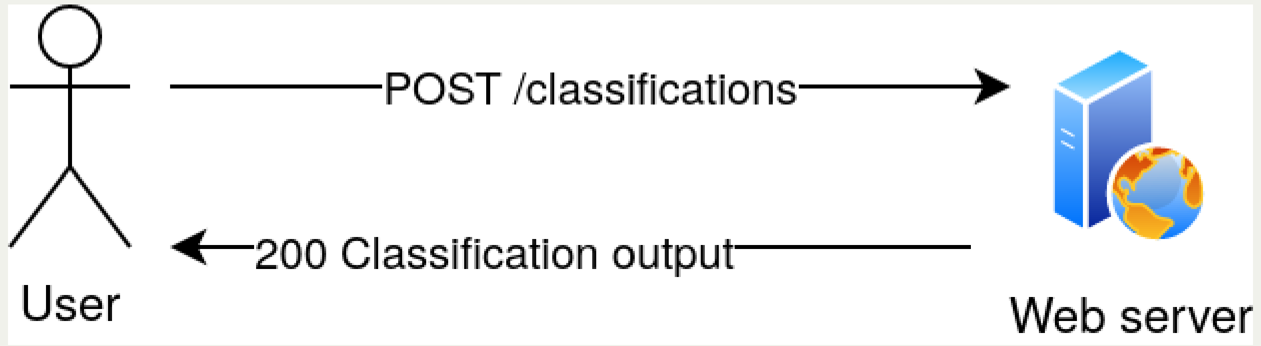
The user should be able to classify an image.



What can the user change? What is fixed?

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...

Modern ML systems are very fast but...

What if classifying the image takes longer?

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.

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.

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.

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.

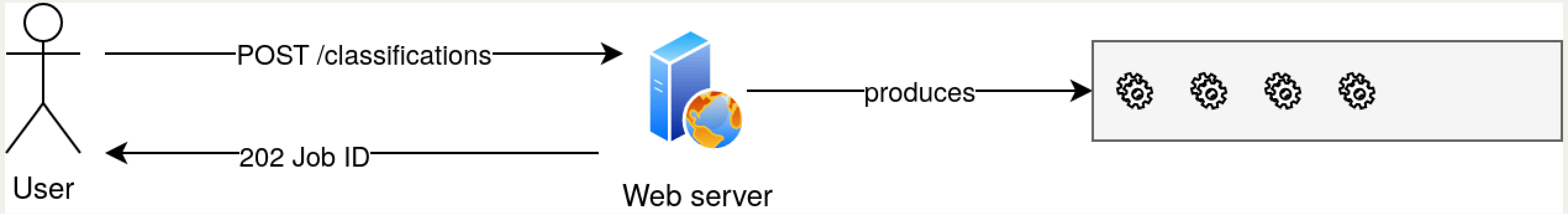
We want to avoid that.

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.

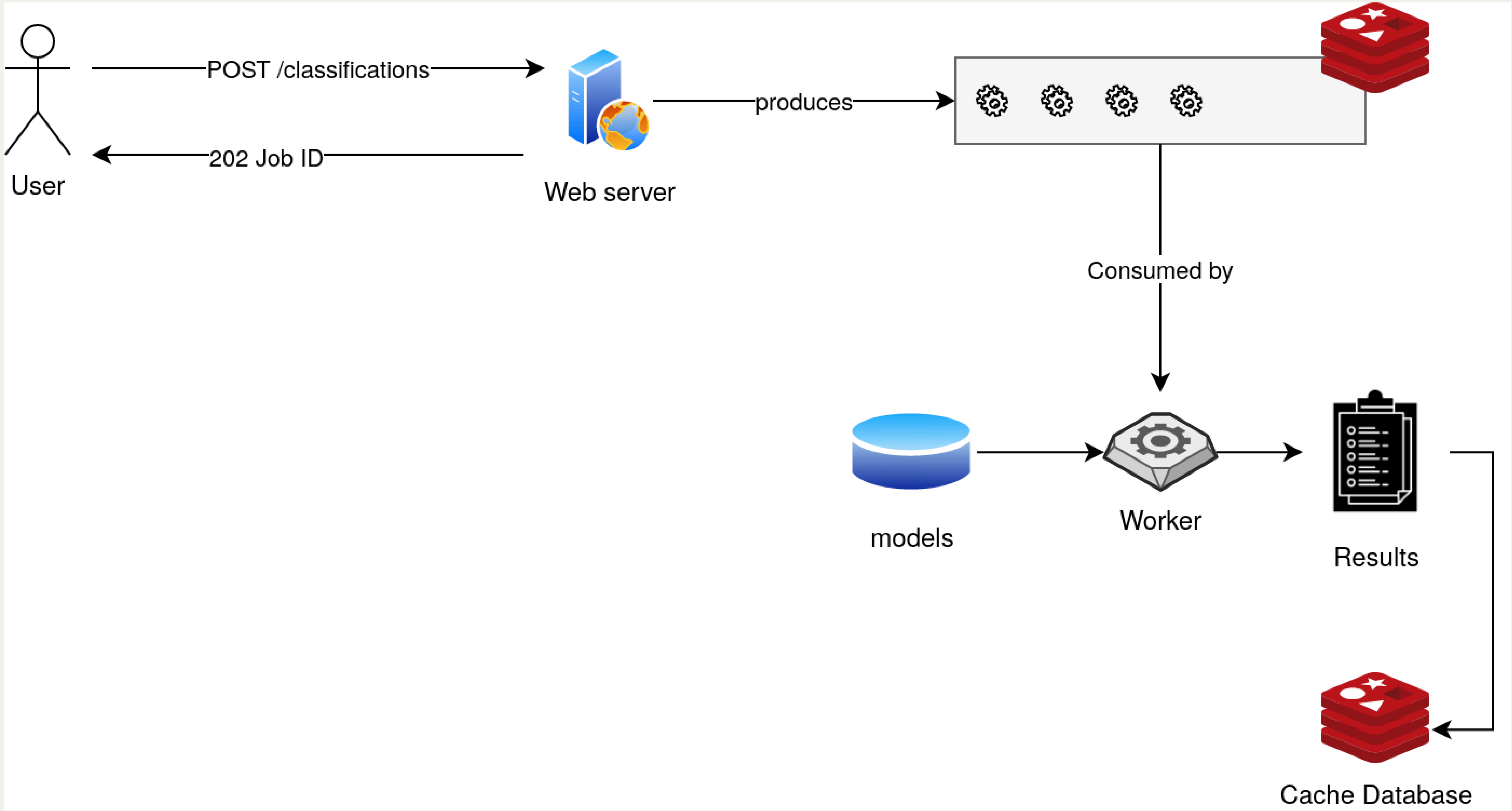
We will implement **asyncronous** jobs.

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.

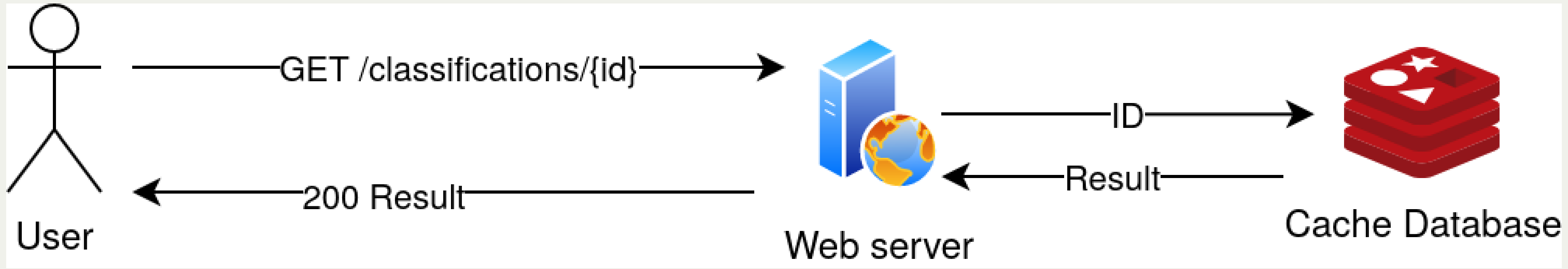


The **worker**, another service of our webserver, takes the enqueued jobs with a **FIFO** (First-In-First-Out) schedule, and processes the requests.

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

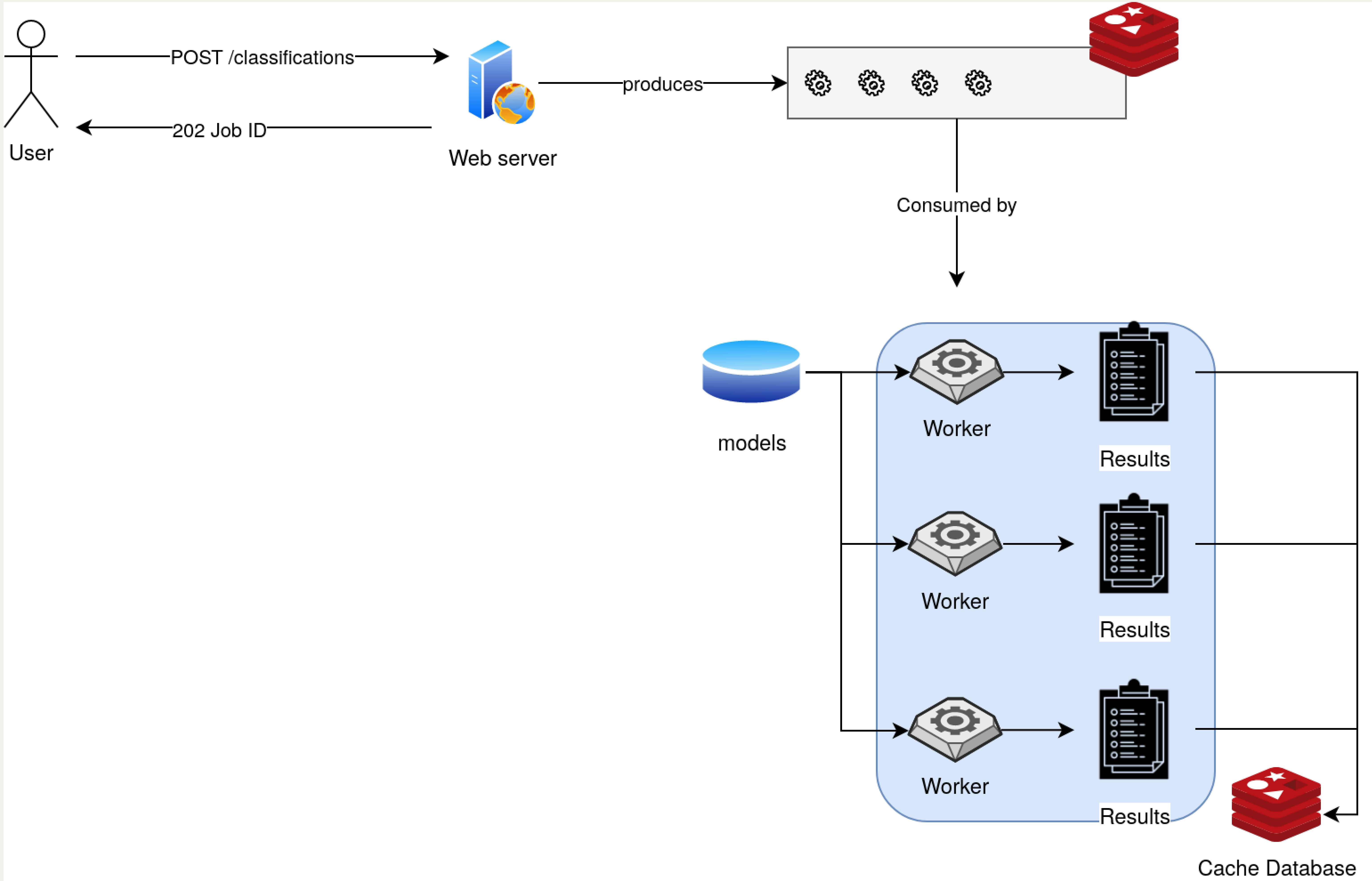

User
POST /classifications
202 Job ID
Web server
produces
Consumed by
models
Worker
Results
Cache Database

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

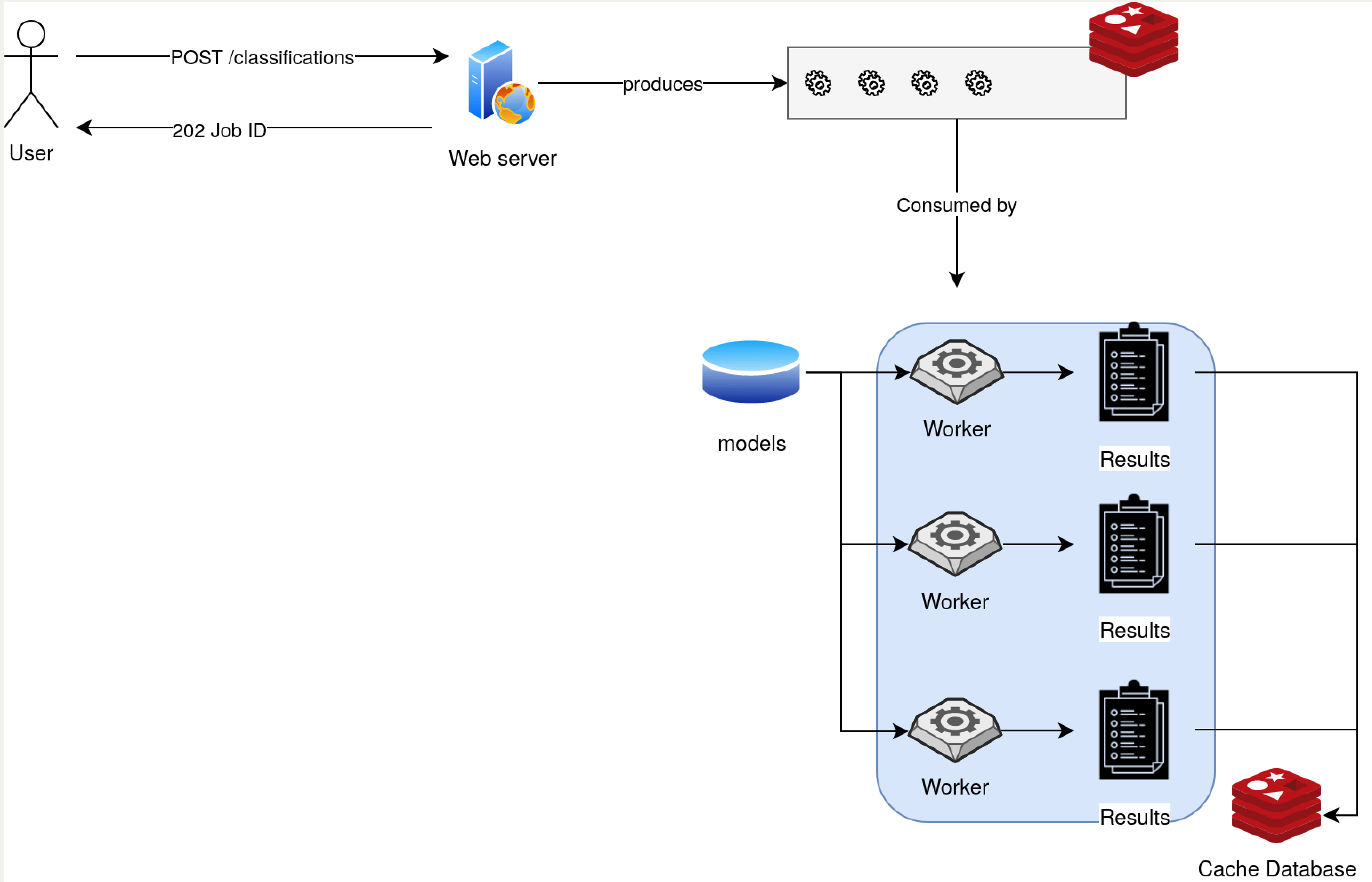

User
POST /classifications
202 Job ID
Web server
produces
Consumed by
models
Worker
Results
Worker
Results
Worker
Results
Cache Database

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

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!

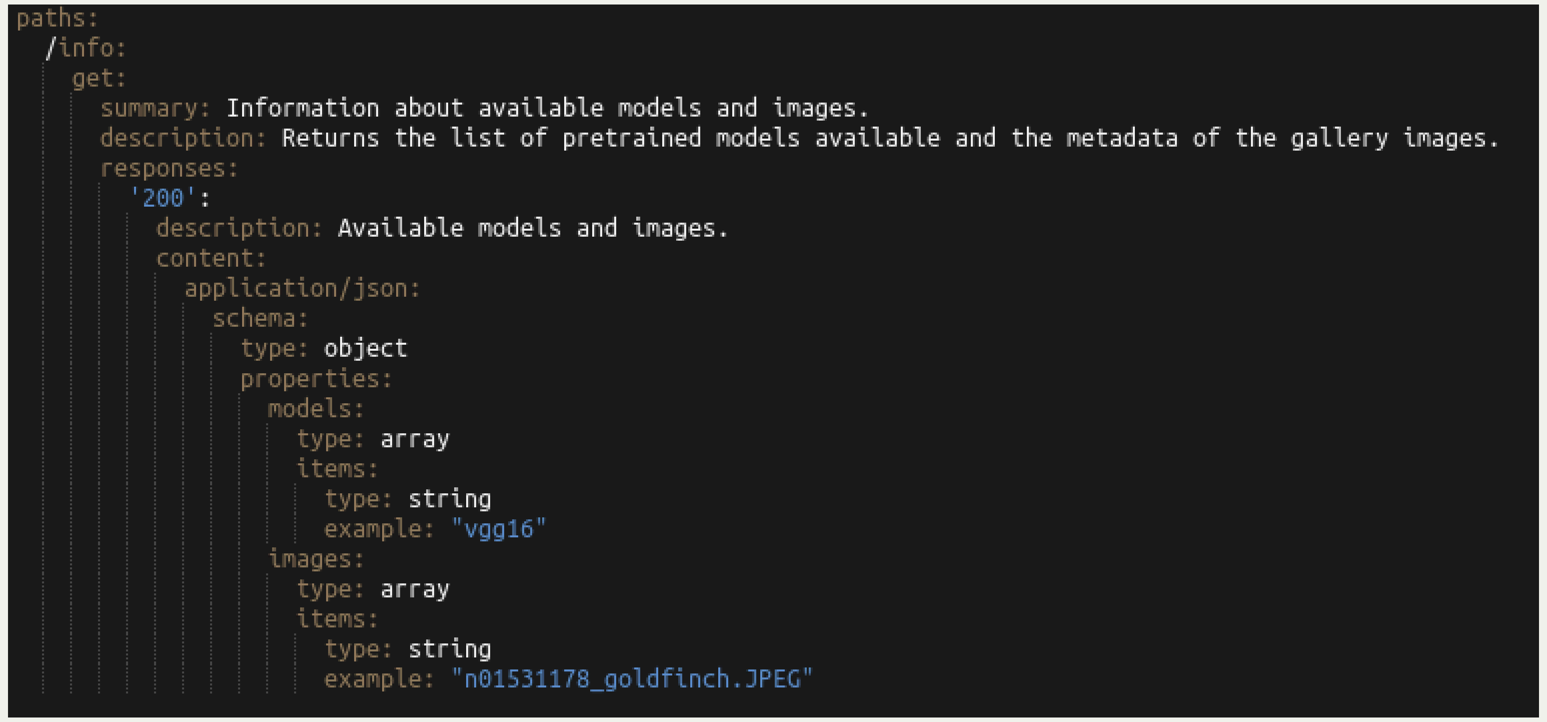
Now we can introduce tools can help us design and maintain our code.

...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:



```yaml
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 later.

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

Now we can start creating our building blocks

Building blocks

Building blocks

- a web server

Building blocks

- a web server
- a queue

Building blocks

- a web server
- a queue
- some worker

Building blocks

- a web server
- a queue
- some worker

To achieve scalability:

- a "box"

Building blocks

- a web server
- a queue
- some worker

To achieve scalability:

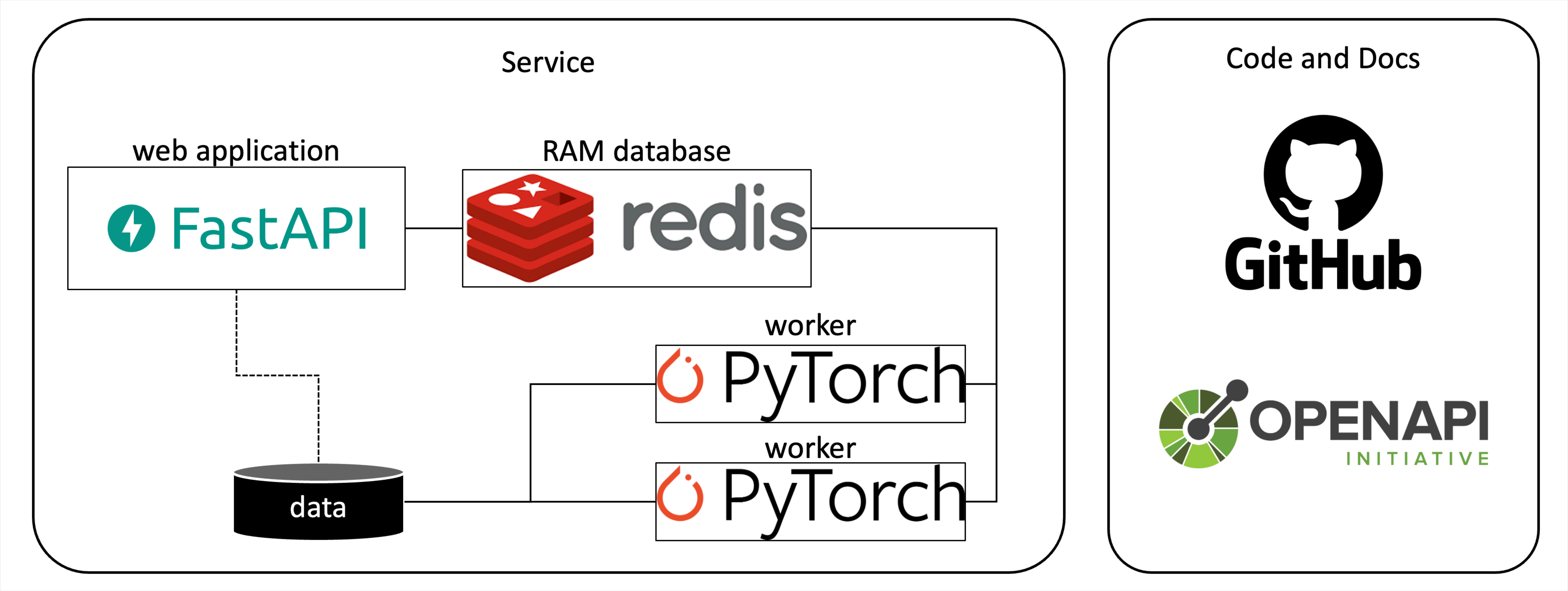
- a "box"
- some storage

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)


Service
web application
FastAPI
RAM database
redis
worker
PyTorch
worker
PyTorch
data
Code and Docs
GitHub
OPENAPI
INITIATIVE

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!