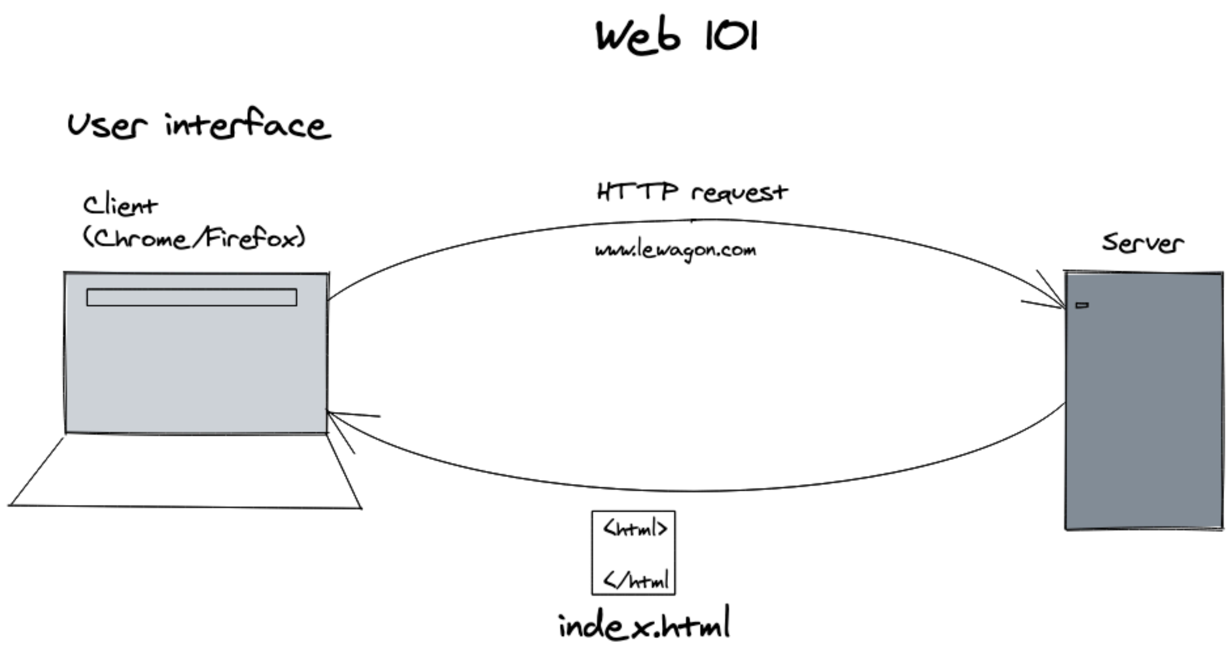
Predict in Production

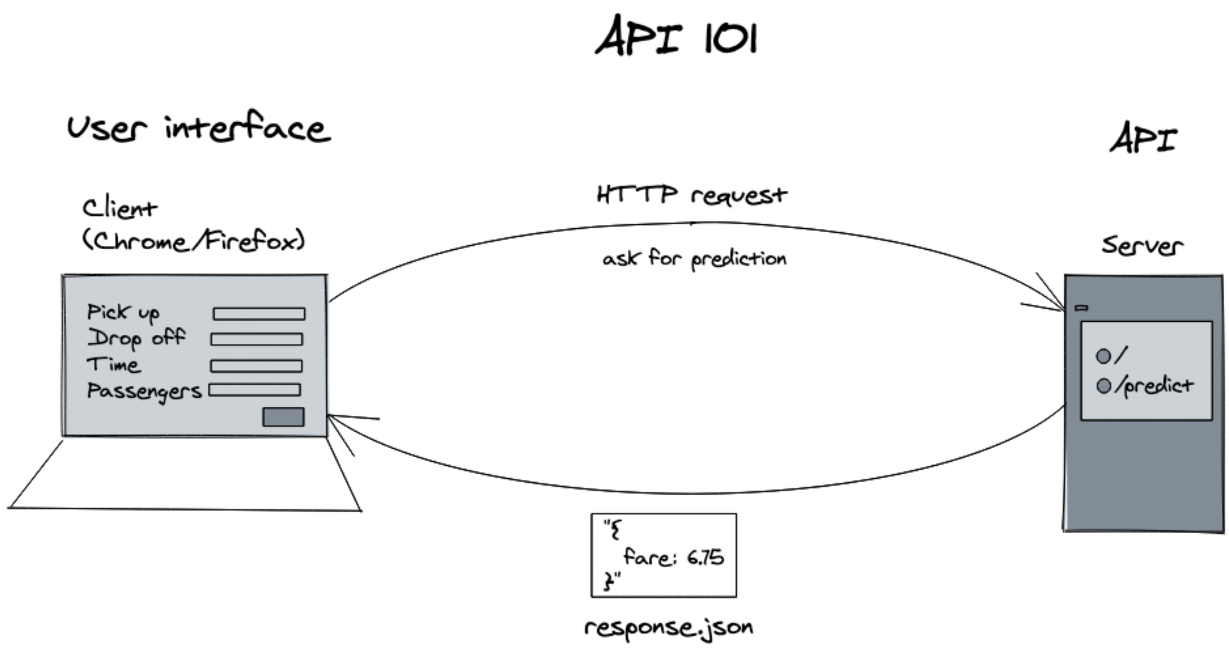
Reminders from Automate Model Lifecycle

Web 101

How Does the Web Work?



How Does an API Work?



E.g. GitHub's interface for humans vs their API

Let's Expose Our Model to the World 🌍

How can we share our model?

Remember the beginning of the bootcamp? We played with the StarWars and Breaking Bad APIs.

Now, we are going to create an API of our own in order to expose our model to other developers.

👉 To achieve that, we will use FastAPI

FastAPI

FastAPI

👉 High-performance Python framework

👉 Easy to learn, fast to code

👉 Automatically generated documentation allows for easy testing of the API endpoints 🎉

```
pip install fastapi
```

ℹ️ [FastAPI documentation]

Root Entry Point

FastAPI uses Python decorators to link the routes that developers will query to the code of the endpoints.

The code of the decorated function will be called whenever an HTTP request is received. The response will be returned to the client as a `JSON` object.

```python
from fastapi import FastAPI

app = FastAPI()

# Define a root `/` endpoint
@app.get('/')
def index():
    return {'ok': True}
```

What if we run the code?

```
python -m simple
```

Nothing happens... 🤔

We need a web server that can

1. listen to HTTP requests being sent to the API
2. call the code of the corresponding endpoint!

👉 We will use Uvicorn

Uvicorn

👉 Lightning fast web server for Python

👉 **Uvicorn** listens to all HTTP requests and calls the function decorated with the corresponding **FastAPI** endpoint.
pip install uvicorn

ℹ️ [Uvicorn documentation]()

Let's run our API using the web server

Uvicorn requires the following parameters:

- the name of the Python file to run (here, `simple`.py)
- the name of the variable containing the **FastAPI** instance (here, `app`)

```
# filename:variable
uvicorn simple:app --reload
```

Now we can browse to the root page of the API: http://localhost:8000/

Documentation and Tests

FastAPI provides automatically generated documentation, allowing developers to simplify their integration of the API. The endpoints of the API can be easily tested through dedicated pages 🎉

Swagger documentation and tests:

- http://localhost:8000/docs

Redoc documentation:

- http://localhost:8000/redoc

👉 The `/docs` endpoint is powered by **Swagger** and comes in very handy when testing our API and verifying that everything is working correctly. It is also very useful for developers wanting to test our API.

Prediction API Use Case

Ask for Prediction

We want to build an API so we can ask it for a prediction.

For example, how long (in minutes) is the line at the Louvre museum for a given day and time. To achieve this, our API should be able to accept an HTTP request with params:

```
url = 'http://localhost:8000/predict'

params = {
  'day_of_week': 0, # 0 for Sunday, 1 for Monday, ...
  'time': '14:00'
}

response = requests.get(url, params=params)
response.json() #=> {wait: 64}
```

The `requests.get(url, params=params)` results in the HTTP request:

👉 `http://localhost:8000/predict?day_of_week=0&time=14:00`

ℹ️ `?day_of_week=0&time=14:00` is called a query string.

`/predict` Endpoint

Let's add a `/predict` endpoint to our API

```python
@app.get('/predict')
def predict():
    return {'wait': 64}
```

Query Parameters

❓What if you want to pass parameters to the endpoint?
FastAPI provides a [simple way]() to do so. You just need need to define the parameters you want to pass as the function parameters.

```python
@app.get('/predict')
def predict(day_of_week, time):
    # Compute `wait_prediction` from `day_of_week` and `time`
    wait_prediction = int(day_of_week) * int(time)

    return {'wait': wait_prediction}
```

⚠️ Query parameters are all `str` so you will need to deal with their conversions into the suitable data types!

What now?

We want to allow every developer to interact with our program, from anywhere in the world.

Now that we are able to run our API on our machine, how can we push it to production?

Platform Types

Choosing a Production Medium

Let's make our models available to the world 🌍

We will provide our predictions through:

- an API, for developers 🤖

- a website, for regular users 🙂 (teaser for the next lecture)

Serving Our Predictions

We need:

- a web server to respond to the API calls
- one or more machines to run that web server
- a reliable and scalable solution

Our Choices

❌ Our machine:

- Is not visible on the internet, and is not up 24/7/365

❌ Google Colab & Vertex AI:

- Do not allow you to run web servers

❌ On-premise:

- We do not want to handle the infrastructure (the network and the hardware)

❌ Infrastructure as a Service (IaaS):

- We do not want to set up the platform (operating system) and the environment (Python)

❌ Platform as a Service (PaaS):

- We need to serve packages containing models that are > 1GB

Best of Both Worlds

👉 We want the flexibility of IaaS

👉 Without having to bother with the complexity of its configuration

Solution: **containers** 🚀

Docker

👆 *This is Moby!*

👉 De facto standard for handling **containers** in production

👉 Allows you to create matching environments for development, testing, QA, and production

👉 Leverages [OS-level virtualization]

What Are Containers?

A **container** is an encapsulated environment that runs an application. Essentially, it's a self-contained computer running Linux!

Running our code inside of a container allows us to deploy it on any machine, without having to know anything about the machine.

Inside of a container, we will find:

👉 The running **code** of our application (a prediction API, for example)

👉 Its **environment** (Python, the required packages, and the Uvicorn server that serves the API)

👉 The **platform** it runs on (the operating system, Linux)

How are containers built?

👉 First, we write a Dockerfile for our application

The **Dockerfile** is a **blueprint** that describes the steps required to create a Docker image.

👉 Then, from the Dockerfile we build a Docker image

The **Docker image** is a **mold** from which containers are created. The Docker image bundles together the code of our application, its environment, and the platform required to run it.

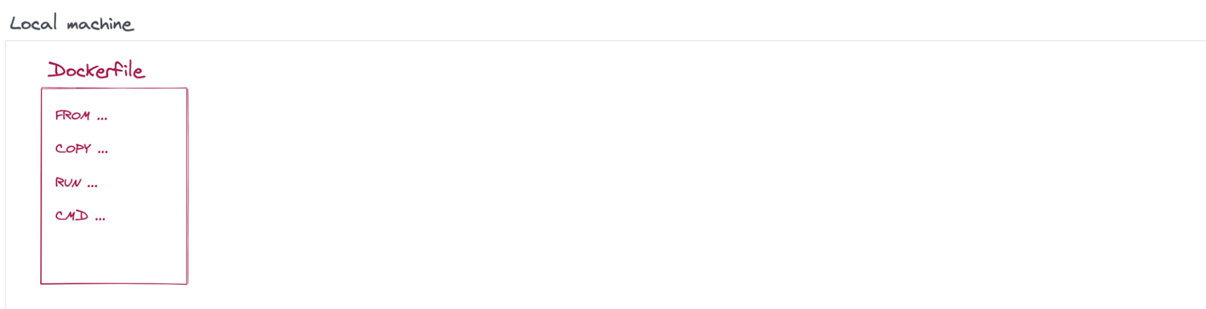
👉 Finally, from the Docker image we can deploy (instantiate) as many containers as we wish. Each container will host a running copy of our application.

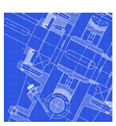
The **container** is a running **instance** of our code.

Dockerfile

The **Dockerfile** is a file that lists all the commands that are needed to build a Docker image.

You can see the Dockerfile as a **blueprint** for a layer cake mold.

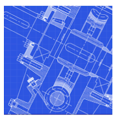




Docker Image

The Docker **image** holds all the files required to instantiate a Docker **container**.

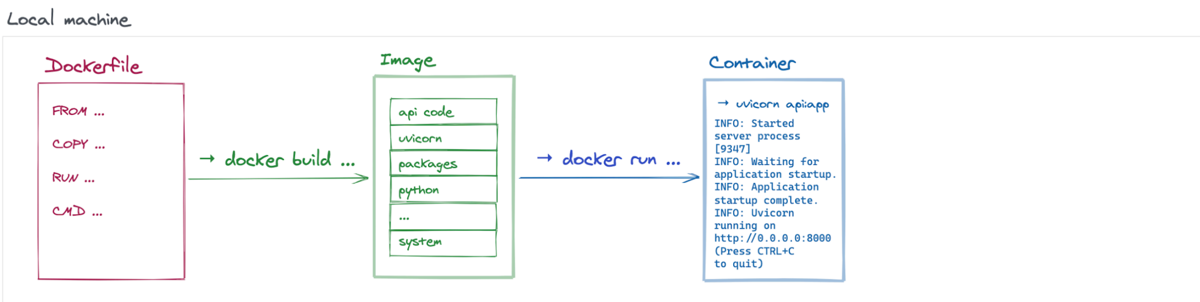
You can see the Docker image as a layer cake **mold**.

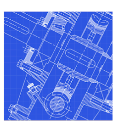


Docker Container

The Docker **container** hosts a running instance of your code.

You can see the Docker container as a **layer cake**.

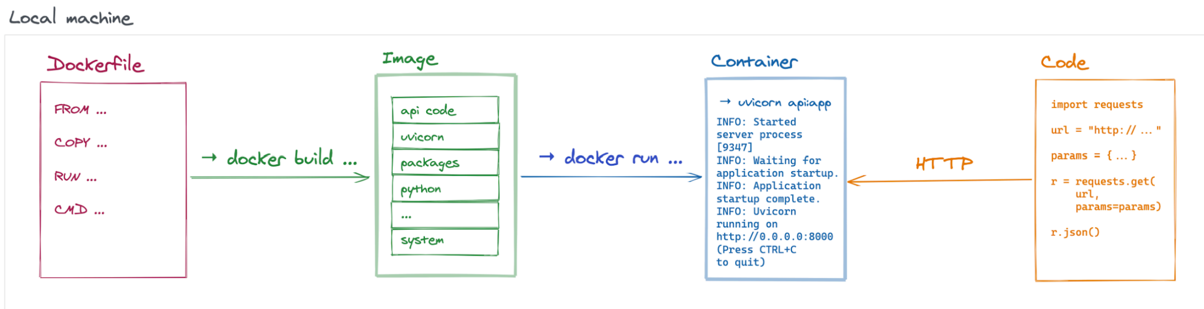


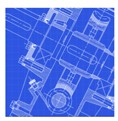


Consumer

Our application will be consumed in different ways depending on its nature. A website will be consumed through a browser, whereas an API will be consumed through code.

You can see the consumer of the application as someone receiving a **piece of the layer cake**.






Create a Docker Image

FROM instruction

Using the `FROM` instruction, we can select the base **layer** of our image. The base layer, which is an image itself, can range from a naked operating system to a fully configured platform.

[Docker Hub](https://hub.docker.com) hosts the base images maintained by the community. It is the GitHub of Docker images.
Let's search for [Python images](https://hub.docker.com).
FROM python:3.10.6-buster

Let's build our first image

1. Build a Docker image

👉 based on the Dockerfile placed in the current directory `.`
docker build .

2. List the Docker images on the machine
docker images

3. Run the image

👉 See how by default the name of the image corresponds to the `FROM` instruction
docker run python:3.10.6-buster

👉 nothing happens... 🤔

4. Let's run our image interactively (`-it`) and ask it to start a shell session (`sh`)

👉 This way we can inspect the content of our container 🔎
docker run -it python:3.10.6-buster sh

We get a shell running **inside** of the container 🤯

We can run any shell command here. For example, let's check the installed Python version:
ls -la
python --version

👉 But our app is nowhere to be seen 🤔

Let's exit the container:
exit

COPY instruction

Once the base layer of the image has been specified, the `COPY` instruction allows us to fill the image with content.
COPY app /app
COPY requirements.txt /requirements.txt

requirements.txt:
fastapi
uvicorn

Let's name our image for convenience

1. Build our new image

👉 name it api with `-t api`
docker build -t api .

2. List all images

👉 see how the new image now has a different `IMAGE ID` from the previous image we had built

3. Run it interactively

```
docker run -it api sh
```

4. Let's see how our app is doing

```
ls -la
cat app/simple.py
cat requirements.txt
```

👉 see how the files addressed by the `COPY` instruction are now inside of the container

```
pip freeze
exit
```

👉 yet no sign of our package on the machine 🤔

RUN instruction

The `RUN` instruction allows us to specify commands that will be executed inside the image. This comes in handy when installing the requirements of our packages.

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

Let's install our packages

1. Build the latest version of the image

👉 you know the drill

2. Run our container interactively

```
# In the shell session
pip freeze
exit
```

3. Everything seems to be ok, let's run our container!

docker run api

👉 aw... 🤔

CMD instruction

The `CMD` instruction is the last instruction of a Dockerfile. It allows us to specify which command the container should run once it has started.

CMD uvicorn app.simple:app --host 0.0.0.0

⚠️ The `--host 0.0.0.0` **flag** tells Uvicorn to listen to all the network connections inside the container. Without this parameter, you will not be able to reach your API through Uvicorn.

Finally, let's connect to our running container

1. Run the container

docker run -p 8080:8000 api *# ⚠️ mind the order of these parameters, or you will get a weird error*

The `-p 8080:8000` **flag** maps the 8080 port on your machine to the 8000 port inside the container.

❓Why 8080? The value does not matter, as long as it is within the [0, 65535] range, and as long as no other application on your machine is already using it.

❓Why 8000? It is the default port used by Uvicorn (you may change it with the `--port 1234`flag).

2. Connect to our API

http://localhost:8080/

How to Stop a Container

1. List running containers

docker ps

2. Stop the image

👉 use the correct CONTAINER ID

docker stop 152e5b79177b

3. If you are in a hurry

👉 use with caution, only if the image refuses to stop
docker kill 152e5b79177b

Artifact Registry

Google Artifact Registry is a cloud storage service for Docker images with the purpose of allowing **Cloud Run** or **Kubernetes Engine** to serve them.

It is, in a way, similar to **GitHub** allowing you to store your git repositories in the cloud — except Google Artifact Registry lacks a dedicated user interface and additional services such as `forks`and `pull requests`.

The goal of Artifact Registry is to act as a warehouse. Whenever we ask **Cloud Run** or **Kubernetes Engine** to create a new instance of our code, they fetch the image we want them to use from the Registry. They will then instantiate the image as one (or multiple) container(s) serving our code.

Parameters

In order to run the commands to push our image to Artifact Registry, we first need to define

👉 the **GCP project identifier** of the project under which we will store our image

👉 the **name** for our image, which will identify it and will be visible in Artifact Registry

👉 the **name of the repo** for our Docker image to be stored in

👉 the **GCP region** in which we want our image to be stored and deployed on Cloud Run

GCP Project Identifier

You can retrieve your project ID either from the [Google Cloud Console]() or through the command line:

```
gcloud projects list
```

Once we have our project ID, let's assign it in a `.env` and load with `direnv` so we can use it in the next few commands.

```
GCP_PROJECT_ID="replace-me-with-your-project-id"
```

👉 Make sure that the environment variable is correctly defined

```
echo $GCP_PROJECT_ID
```

Docker Image Name

Now we need to define a name for the image that will be stored in Artifact Registry.

You may use whatever name seems appropriate, we strongly advise using a kebab case identifier.

```
DOCKER_IMAGE_NAME="name-of-my-image-in-kebab-case"
echo $DOCKER_IMAGE_NAME
```

Regions

The region in which our code will be deployed should be coherent with the region in which our image is stored.

If a region is currently configured for the project, it will be listed (as a zone) when using:

```
gcloud config list
```

If not, we can pick a region from the list of available regions. The identifiers for the regions are similar to the ones of the zones, but without the -a, -b and -c suffixes.

We already have been using a region all week, so we'll use the same one for both storing and deploying our model:

```
GCP_REGION="europe-west1"  # replace with the appropriate region or multi-region

echo $GCP_REGION
```

Storing our Docker images

To push our image up to the Artifact Registry, we'll first need a **repository** in which we'll store it. This only requires two things:

1) We need to configure Docker to use the Google Cloud CLI to authenticate requests to Artifact Registry. To set up authentication to Docker repositories in the region `"europe-west1"`, run the following command (you'll only need to do this **once**):

```
gcloud auth configure-docker $GCP_REGION-docker.pkg.dev
```

2) Once we've done that, we choose a name for our repository and create it:

```
DOCKER_REPO_NAME="my-first-repo"
echo $DOCKER_REPO_NAME
gcloud artifacts repositories create $DOCKER_REPO_NAME --repository-format=docker \
--location=$GCP_REGION --description="My first repository for storing Docker images in GAR"
`
```

We can check that it's appeared in our GCP 🧐

Updating the Dockerfile for Google Cloud Run

⚠️ Remember that Uvicorn serves on port 8000 by default? Well, **Google Cloud Run** requires that every server running inside of its containers runs on a specific port that is uniquely designated to each server, and is defined by them via the `$PORT` environment variable. This allows Cloud Run to monitor the code and restart the container if the server crashes.

👉 In order to do that, we need to update our image so that Uvicorn listens to the `$PORT` by using the `--port` parameter

```
FROM python:3.8.6-buster

COPY api /api
COPY project /project
COPY model.joblib /model.joblib
```

COPY requirements.txt /requirements.txt

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

CMD uvicorn api.simple:app --host 0.0.0.0 --port $PORT

Build our Image for Artifact Registry

Once we have updated the Dockerfile to use the $PORT defined by **Cloud Run**, we can build our image one last time, this time tagging it with all of our environment variables and a **version number**(e.g. `0.1`):
docker build -t
$GCP_REGION-docker.pkg.dev/$GCP_PROJECT_ID/$DOCKER_REPO_NAME/$DOCKER_IMAGE_NAME:0.1 .

We want to ensure that our modifications are working correctly, but in order for our Dockerfile to work correctly, we need to define the $PORT environment variable ourselves by using `-e PORT=8000`:
docker run -e PORT=8000 -p 8080:8000
$GCP_REGION-docker.pkg.dev/$GCP_PROJECT_ID/$DOCKER_REPO_NAME/$DOCKER_IMAGE_NAME:0.1

👉 The sooner you detect an error in the code or in the configuration of the Docker project, the less time you lose.

👉 Let's verify that everything is OK on http://localhost:8080/

Side note for M1/M2 users

- When using an M1 or M2 system, Docker will create `linux/arm64` images by default
- These work only on machines that are also using ARM architecture
- But most machines (including those on GCP) use an x64 architecture

To adjust for this, we can build a specific image for running on GCP by adding the `--platform`flag:

```
docker build --platform linux/amd64 -t
$GCP_REGION-docker.pkg.dev/$GCP_PROJECT_ID/$DOCKER_REPO_NAME/$DOCKER_IMAGE_NAME:0.1 .
```

Push our Image to Artifact Registry

Finally, we can push the image to **Artifact Registry**:
docker push
$GCP_REGION-docker.pkg.dev/$GCP_PROJECT_ID/$DOCKER_REPO_NAME/$DOCKER_IMAGE_NAME:0.1

Cloud Run

Cloud Run is the Google service dedicated to serving Docker images.

Deploy our Image to Cloud Run

Once our image is stored on Artifact Registry, deploying our code to Cloud Run is easy:
gcloud run deploy --image
$GCP_REGION-docker.pkg.dev/$GCP_PROJECT_ID/$DOCKER_REPO_NAME/$DOCKER_IMAGE_NAME:0.1 --region $GCP_REGION

Bibliography

- Sanitize query parameters with Pydantic typehints
- `CMD` vs `ENTRYPOINT`

Your turn! 🚀