Lab’s

✅ Lab 1: *Coding with AI*

Theory:

Roo Code - is an AI-powered autonomous coding agent that lives in your editor. It integrates with various AI models and providers, allowing you to connect to services like OpenAI, Google Gemini, Anthropic, OpenRouter, and local LLMs via platforms like LM Studio or Ollama.

Flask is a minimal, open-source web framework written in Python. It allows developers to create web applications, REST APIs, and backend services without requiring a large, complex setup. It follows the “micro-framework” philosophy, meaning it provides only the core features needed to run a web application, and you add additional components (database, authentication, templates, etc.) as needed.

Key characteristics of Flask:

1. **Lightweight and flexible:** Flask does not impose a specific project structure. You decide how to organize files, which libraries to use, and how the application grows.
2. **Built-in development server and debugger:** Flask includes a simple server and debug mode to help test applications quickly during development.
3. **Routing system:** You define URL endpoints using Python functions. For example:

```
@app.route("/hello")
def home():
    return "Hello World"
```

4. **Extensible with many plugins:** You can add libraries such as:
    - Flask-SQLAlchemy (databases)

- Flask-Login (authentication)
- Flask-RESTful (API support)
- Flask-Migrate (database migrations)

5. **Common use cases:** Web applications, REST and JSON APIs, Microservices.

Example:

Python code to Generate HTTP Response to: `http://<service_host>/hello`

Coding example

```python
from flask import Flask, jsonify

app = Flask(__name__, instance_relative_config=True)

@app.route('/hello', methods=['GET'])
def hello() -> Any:
    return jsonify(message="Hello, World!")

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

- **Route:** specifies the path `(/hello)` and can specify the HTTP method `(default = GET)`. It is a Flask annotation.

  A route is and endpoint part of the REST API of our service. `http://<service_host>` `/hello`
- The annotated function `def(hello)` is called when arrives a request for the route.
- A **JSON** containing the message is generated and sent back to the requester.

How to Run Flask Service: From the terminal (inside an activated venv):

1. Install the service requirements: `pip install flask`
2. Run the service: `flask run —host=0.0.0.0 —port=5005`

   Now the service will be waiting on any network interface (0.0.0.0) at port 5005. The terminal will log the events of the service, to close the service and take control of the terminal ctrl+C (Command+C).

If the 5005 port is used by another service of your system, you can change it.

3. Open a browser and visit `http://localhost:5005/hello`. You should see the result of the execution of the code. By default, you have to call the Python file app.py.

Lab:

Objective

The goal of this laboratory work was to create a simple **Calculator API** using **Flask** and **AI coding tools**.

The exercise demonstrates how to set up an AI-assisted development environment and use it to generate functional backend code.

Tools and Technologies

- **Python 3.12** — programming language.
- **Flask** — lightweight web framework for creating APIs.
- **Visual Studio Code (VS Code)** — code editor.
- **AI Tools:** GitHub Copilot or Roo Code (optional).
- **Operating System:** Windows / Linux / macOS.

Step 1 — Environment Setup

After installing all the required software (Python, VS Code, and Flask),

I created a new folder named `lab1_flask_ai` for this laboratory project and opened it in VS Code.

From the provided course materials, I copied the file `app.py` into this folder.

The file contained an example Flask web service with a `/hello` route that returned a JSON response.

Step 2 — Creating a Virtual Environment

To isolate dependencies, I created a virtual environment inside the project folder using:

```
python -m venv venv
```

Then activated it:

```
source venv/bin/activate     # for macOS/Linux
```

Step 3 — Installing Flask

With the environment activated, I installed Flask:

```
pip install flask
```

This allowed me to run the base service using:

```
flask run --host=0.0.0.0 --port=5005
```

When visiting http://localhost:5005/hello,

the Flask service responded with a JSON message, confirming the setup was successful.

Step 4 — Generating the Calculator API

The task was to extend the base Flask application by creating an API that performs basic arithmetic operations.

Using an AI tool (or manually copying the generated code), I implemented four API endpoints:

- `/add?a=...&b=...`
- `/sub?a=...&b=...`
- `/mul?a=...&b=...`
- `/div?a=...&b=...`

Each route:

1. Receives two query parameters `a` and `b`.
2. Performs the corresponding mathematical operation.
3. Returns the result as a JSON object.

Step 5 — AI-Generated Flask Code

The following code was generated with the help of AI:

```
from flask import Flask, request, jsonify

app = Flask(__name__)

@app.route('/add')
def add():
    try:
        a = float(request.args.get('a'))
        b = float(request.args.get('b'))
        return jsonify({"operation": "add", "result": a + b})
    except (TypeError, ValueError):
        return jsonify({"error": "Invalid input. Use /add?a=10&b=5"}), 400

@app.route('/sub')
def sub():
    try:
        a = float(request.args.get('a'))
        b = float(request.args.get('b'))
        return jsonify({"operation": "sub", "result": a - b})
    except (TypeError, ValueError):
        return jsonify({"error": "Invalid input. Use /sub?a=10&b=5"}), 400

@app.route('/mul')
def mul():
    try:
        a = float(request.args.get('a'))
        b = float(request.args.get('b'))
        return jsonify({"operation": "mul", "result": a * b})
    except (TypeError, ValueError):
        return jsonify({"error": "Invalid input. Use /mul?a=10&b=5"}), 400

@app.route('/div')
def div():
    try:
        a = float(request.args.get('a'))
        b = float(request.args.get('b'))
        if b == 0:
            return jsonify({"error": "Division by zero is not allowed"}), 400
```

```
        return jsonify({"operation": "div", "result": a / b})
    except (TypeError, ValueError):
        return jsonify({"error": "Invalid input. Use /div?a=10&b=5"}), 400
```

Step 6 — Testing the Flask Service

To verify functionality, I ran the service:

```
flask run --host=0.0.0.0 --port=5005
```

Then tested endpoints using the browser or `curl`:

```
curl "http://localhost:5005/add?a=10&b=5"
curl "http://localhost:5005/mul?a=3&b=7"
```

Expected outputs:

```
{"operation": "add", "result": 15.0}
{"operation": "mul", "result": 21.0}
```

This confirmed the service was running correctly and performing the required operations.

Conclusion

In this lab, I successfully:

- Set up a Python Flask environment.
- Used AI to generate and understand backend service code.
- Tested a REST API that performs calculator operations.

This exercise demonstrated how AI can accelerate software development while helping understand core web concepts such as **APIs**, **routes**, and **HTTP requests**.

Code Explanation

`from flask import Flask, request, jsonify`

- This imports tools from Flask:

- **Flask** → creates the web app
- **request** → lets you read data from the URL (like `a=10&b=5`)
- **jsonify** → sends data back in JSON format (like `{"result": 15}`)

`app = Flask(__name__)`

- Creates the main app (website).
- You must do this to start a Flask application.

`@app.route('/add')` This tells Flask: "When someone visits `/add`, run the function below."

`def add():` This is the function that will run when visiting `/add`.

`try:`

- Try to run the code inside.
- If something goes wrong (like invalid input), go to `except`.

```
a = float(request.args.get('a'))
b = float(request.args.get('b'))
```

- Reads the numbers from the URL:
  - `a = ?`
  - `b = ?`
- Example: `/add?a=10&b=5`
- Converts them to `float` (numbers with decimals).

`return jsonify({"operation": "add", "result": a + b})`

- If everything is correct, return JSON:
  - `"operation": "add"`
  - `"result": a + b` (the sum)

`except (TypeError, ValueError):`

- If `a` or `b` is missing or not a number, this block catches the error.

`return jsonify({"error": "Invalid input. Use /add?a=10&b=5"}), 400`

- Sends an error message with status **400** (bad request).

Route: /sub

Same structure as `/add`, but: It subtracts instead of adding.

```
return jsonify({"operation": "sub", "result": a - b})
```

Route: /mul

Same thing again but: Multiplies.

```
return jsonify({"operation": "mul", "result": a * b})
```

Route: /div

Again similar, but with special logic for division: Checks if the second number is **0**. If yes → returns an error about division by zero.

```
if b == 0:
 return jsonify({"error": "Division by zero is not allowed"}), 400
```

Otherwise:

```
return jsonify({"operation": "div", "result": a / b})
```

Divides `a` by `b`.

Home Page Route: /

```
@app.route('/')
def home():
```

- When someone enters the main page (`/`), show a welcome message.

```
return jsonify({
   "message": "Welcome to the AI-powered Calculator!",
   "routes": ["/add?a=&b=", "/sub?a=&b=", "/mul?a=&b=", "/div?a=&b="]
  })
```

- Shows:
  - A message
  - A list of available calculator endpoints

This is basically a “homepage” for your tiny calculator API.

Running the App

```
if __name__ == '__main__':
```

- Means: “Only run the app if this file is executed directly.”

```
app.run(host='0.0.0.0', port=5005, debug=True)
```

- **host='0.0.0.0'** → makes the app accessible from your network

- **port=5005** → your app will run on port 5005
- **debug=True** → helpful for development; shows errors in the browser

✅ Lab 2: User Stories

Theory:

User stories are short, clear descriptions of a feature written from the perspective of the end user. They are used in Agile software development to define what the user needs and why.

User stories do not describe technical details. They describe what should be built, not how to build it.

Structure (standard format):

- **As a** <role> **I want to** <action>, **So that** <value>

Purpose:

- Understand user needs.
- Guide design and development.
- Help teams estimate effort.
- Focus on delivering user value.

Lab:

Created a 10 user stories for player, and putted them in relevance from the most important and to the least important.

- **As a** <role> **I want to** <action>, **So that** <value>

User Storie's:

#	User Story	Reason / Importance
1	🔐 **As a player, I want to authorize, so that I can access my profile.**	Authorization is the **entry point** — required before any interaction.
2	👤 **As a player, I want to have a profile, so that I can see my**	After login, the player needs a **profile** to track identity and stats.

#	User Story	Reason / Importance
	and others' stats.	
3	⚔️ **As a player, I want to choose an opponent, so that I can invite him.**	Once logged in, selecting an opponent is the **first step to starting a game.**
4	🃏 **As a player, I want to choose a sequence of cards, so that I can have a strategy.**	Pre-battle setup — defining your card strategy before the game starts.
5	🎯 **As a player, I want to select a card, so that I can put it into battle.**	The **core action** of gameplay — essential for the main interaction.
6	🧱 **As a player, I want to have a scoreline while playing, so that I can see the score.**	Real-time feedback during the match to guide decisions.
7	⏳ **As a player, I want to see how many rounds we have left, so that I can think about my turns.**	Helps players plan strategically mid-game.
8	🏁 **As a player, I want to see the final score, so that I can know who won.**	Core post-game feedback — completes the match loop.
9	🚪 **As a player, I want to leave the game, so that I can stop the game anytime.**	Optional control feature — not core to flow, but important for UX.
10	🏆 **As a player, I want to have achievements, so that I can show them in my profile.**	A **reward/retention** feature — enhances engagement but not required for MVP.

✅ Lab 3: REST and OpenAPI

Theory:

API - Application Programming Interface, its a way for 2 computers to talk to each other. Common standard used by most applications to talk to the servers is REST API.

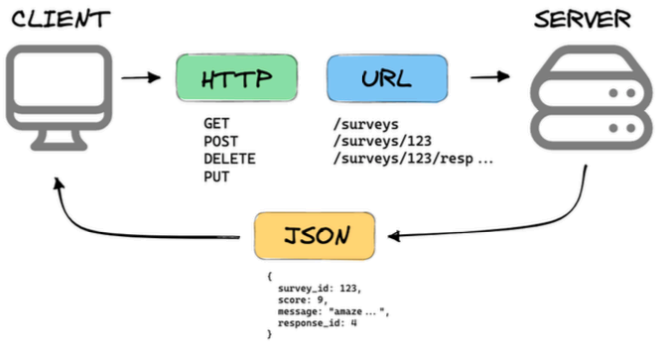
REST is most common communication standard between computers over Internet.

REST (REpresentational State Transfer): This is an architectural style for designing networked applications. The key idea is to treat everything as

a **resource** (a "noun," like `/users` or `/orders`).

Representations (usually JSON) transfer state between client and server.

Endpoint: URL + HTTP method



- **HTTP Methods (Verbs):** You interact with resources using standard HTTP methods, which have specific meanings:
  - `GET`: Read a resource (e.g., get a user's details).
  - `POST`: Create a new resource (e.g., create a new user) or trigger an action.
  - `PUT`: Replace/update a resource completely.
  - `DELETE`: Remove a resource.
- **RESTful Path Design:** Paths should be "nouns" (plural, lowercase) like `/courses` or `/assignments`. Avoid "verbs" in paths like `/create-course`.
- **Parameters:**
  - **Path Parameters:** Identify specific resource (e.g., `/courses/{courseId}`).
  - **Query Parameters:** Filter or modify a request (e.g., `/orders?status=shipped`).
- **HTTP Status Codes:** Your API must return standard codes to tell the client what happened.
  - `2xx` **(Success):** `200 OK`, `201 Created`.
  - `3xx` **(Redirects):** `304 Not Modified`
  - `4xx` **(Client Error):** `400 Bad Request` (e.g., missing parameter), `404 Not Found` (resource doesn't exist). etc.

- `5xx` **(Server Error):** `500`, `502`, `503`

OpenAPI Specification: A machine-readable file (in YAML or JSON format) that describes your entire REST API. It lists all endpoints, parameters, responses, and data models.

Other tools can read this file and automatically generate: API documentation, Swagger UI (pretty documentation website).

We need OpenAPI to document APIs clearly, reduce confusion, allow automatic documentation generation, improve collaboration, and make it easy for other developers or systems to understand and use the API.

- **Swagger:** A set of tools for working with OpenAPI. Swagger kind of just reads this yaml/json OpenApi file and can give you documentation website of your REST API, to make it easy to understand and User Interfaced.
  - **Swagger Editor:** An online tool to write and visualize your OpenAPI spec.
  - **Swagger UI:** The web representation you see as a result.

OpenAPI Code Parts:

`openapi: 3.0.3:` This means "we are using OpenAPI version 3.0.3".

---

info: Basic information about your API. This is just documentation metadata.

```
title: AI-Powered Calculator API
version: 1.0.0
description: A simple RESTful API...
```

---

servers: Where your API is hosted. This tells tools like Swagger UI where to send requests.

```
url: http://localhost:5005
```

---

paths: This is the MOST important part. It lists all your API endpoints. Here you have ONE endpoint: `/api/calculate`

---

post: Means this endpoint accepts a **POST request**. This explains what the endpoint does.

summary: Perform a calculation
description: Takes two numbers and an operation...

---

requestBody: What data the client MUST send. It says:

- The body must be JSON
- JSON must contain:
  - `operation` (add, sub, mul, div)
  - `a` (number)
  - `b` (number)

Example is given so developers know how to call the API:

```
{
  "operation": "add",
  "a": 10,
  "b": 5
}
```

`responses:` All possible results your API can return.

✔ **200 — Success** It returns:

```
{
  "operation": "add",
  "a": 10,
  "b": 5,
  "result": 15
}
```

✔ **400 — Bad Request** It returns something like: This teaches users how to handle errors. `{ "error": "Invalid request..." }`

Lab:

Step 1: Changed the code of Lab 1 to the RESTful API code bellow:

```
from flask import Flask, request, jsonify

app = Flask(__name__)

@app.route('/api/calculate', methods=['POST'])
def calculate():
    """
    RESTful endpoint that performs arithmetic operations.
    Expects JSON body with fields: operation, a, b
    Example:
    {
        "operation": "add",
        "a": 10,
        "b": 5
    }
    """
    data = request.get_json()

    # Validate input
    if not data or 'operation' not in data or 'a' not in data or 'b' not in data:
        return jsonify({"error": "Invalid request. Example: {'operation':'add',
'a':10, 'b':5}"}), 400

    operation = data.get('operation')
    a = data.get('a')
    b = data.get('b')

    # Validate numbers
    try:
        a = float(a)
        b = float(b)
    except (TypeError, ValueError):
        return jsonify({"error": "Operands must be numbers."}), 400

    # Perform the requested operation
    if operation == "add":
        result = a + b
    elif operation == "sub":
```

```
        result = a - b
    elif operation == "mul":
        result = a * b
    elif operation == "div":
        if b == 0:
            return jsonify({"error": "Division by zero is not allowed."}), 400
        result = a / b
    else:
        return jsonify({"error": "Unsupported operation. Use one of: add, sub,
mul, div."}), 400

    return jsonify({
        "operation": operation,
        "a": a,
        "b": b,
        "result": result
    }), 200


@app.route('/')
def home():
    return jsonify({
        "message": "Welcome to the RESTful Calculator API!",
        "usage": {
            "POST /api/calculate": {
                "body_example": {"operation": "add", "a": 10, "b": 5}
            }
        }
    })


if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5005, debug=True)
```

To see if it works, again we activate Virtual Environment: `(venv)`

`source venv/bin/activate`

then we run our flask service with:

`flask run --host=0.0.0.0 --port=5005`

so we test it with this curl instead of previous one:

```
curl -X POST http://localhost:5005/api/calculate \
-H "Content-Type: application/json" \
-d '{"operation":"add","a":10,"b":5}'
```

if it works well in shell, we get message like this:

```
127.0.0.1 - - [09/Nov/2025 18:53:54] "GET / HTTP/1.1" 200 -
```

if not then **400 / 404**

Code Explanation: Difference between old and RESTful Code version is that here we use POST requests with JSON bodies. And there is **one Endpoint** for one resource:

```
@app.route('/api/calculate', methods=['POST'])
```

Data is sent in JSON REST APIs usually accept and return JSON as we did.

Old code uses GET requests, Client sends data in the URL: `/add?a=10&b=5`

New code uses POST requests, Client sends data in JSON body: `{"operation": "add", "a": 10, "b": 5}`

Different input validation. Old Code: `request.args.get('a')`

New Code: Extract data using JSON `request.get_json()`

Step 2: Then created **openapi.yaml** file inside the project, Where app.py is located.

Created code OpenAPI:

```
openapi: 3.0.3
info:
  title: AI-Powered Calculator API
  version: 1.0.0
  description: A simple RESTful API built with Flask that performs basic arith
metic operations.

servers:
  - url: http://localhost:5005
    description: Local development server

paths:
```

```
  /api/calculate:
    post:
      summary: Perform a calculation
      description: Takes two numbers and an operation, and returns the resul
t.
      requestBody:
        required: true
        content:
          application/json:
            schema:
              type: object
              properties:
                operation:
                  type: string
                  enum: [add, sub, mul, div]
                  example: add
                a:
                  type: number
                  example: 10
                b:
                  type: number
                  example: 5
      responses:
        '200':
          description: Successful calculation
          content:
            application/json:
              schema:
                type: object
                properties:
                  operation:
                    type: string
                  a:
                    type: number
                  b:
                    type: number
                  result:
                    type: number
```

```
          example:
            operation: add
            a: 10
            b: 5
            result: 15
        '400':
          description: Invalid input or bad request
          content:
            application/json:
              schema:
                type: object
                properties:
                  error:
                    type: string
              example:
                error: "Invalid request. Example: {'operation':'add', 'a':10, 'b':5}"
```

We put this code in the **openapi.yaml** file. To see it's web representation we use Swagger, go to https://editor.swagger.io . And just open our **openapi.yaml** or just put code right there. Remember that our, flask should be running on the moment, when we want to see it.

Code Explanation:

openapi: 3.0.3: This means "we are using OpenAPI version 3.0.3".

---

info: Basic information about your API.

```
title: AI-Powered Calculator API
version: 1.0.0
description: A simple RESTful API...
```

This is just documentation metadata.

---

servers: Where your API is hosted.

```
url: http://localhost:5005
```

This tells tools like Swagger UI where to send requests.

---

paths: This is the MOST important part. It lists all your API endpoints. Here you have ONE endpoint: `/api/calculate`

---

post: Means this endpoint accepts a **POST request**.

```
summary: Perform a calculation
description: Takes two numbers and an operation...
```

This explains what the endpoint does.

---

requestBody: What data the client MUST send. It says:

- The body must be JSON
- JSON must contain:
  - `operation` (add, sub, mul, div)
  - `a` (number)
  - `b` (number)

Example is given so developers know how to call the API:

```
{
  "operation": "add",
  "a": 10,
  "b": 5
}
```

---

responses: All possible results your API can return.

✔️ **200 — Success** It returns:

```
{
  "operation": "add",
  "a": 10,
  "b": 5,
  "result": 15
}
```

✔️ **400 — Bad Request** It returns something like:

```
{
  "error": "Invalid request..."
}
```

This teaches users how to handle errors.

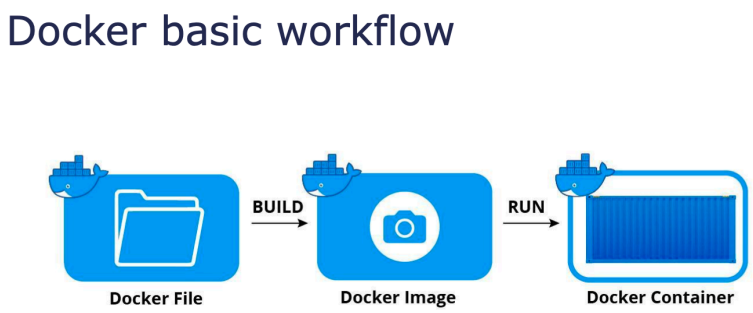
✅ Lab 4: Docker

Theory:

Why use containers?

- **Run anywhere:** same image works on any computer or cloud.
- **Consistent:** includes app + dependencies, no “works on my machine.”
- **Fast & lightweight:** starts in seconds, uses less resources than VMs.
- **Isolated:** each app runs safely in its own space.
- **Scalable:** easy to copy, update, or roll back in deployment.

Docker is a tool that helps developers run applications in a clean, isolated environment called a **container**. It makes sure the application runs the same way on any computer, server, or cloud system.



1. Container: is a lightweight, isolated box that holds:

- The application
- Required libraries and dependencies
- Configuration

It shares the host operating system, so it is fast and efficient. Think of a container like a portable suitcase that always has everything the app needs inside.

It is possible to have multiple containers from an image.

---

2. Image: is a blueprint or recipe for creating a container.

Example:

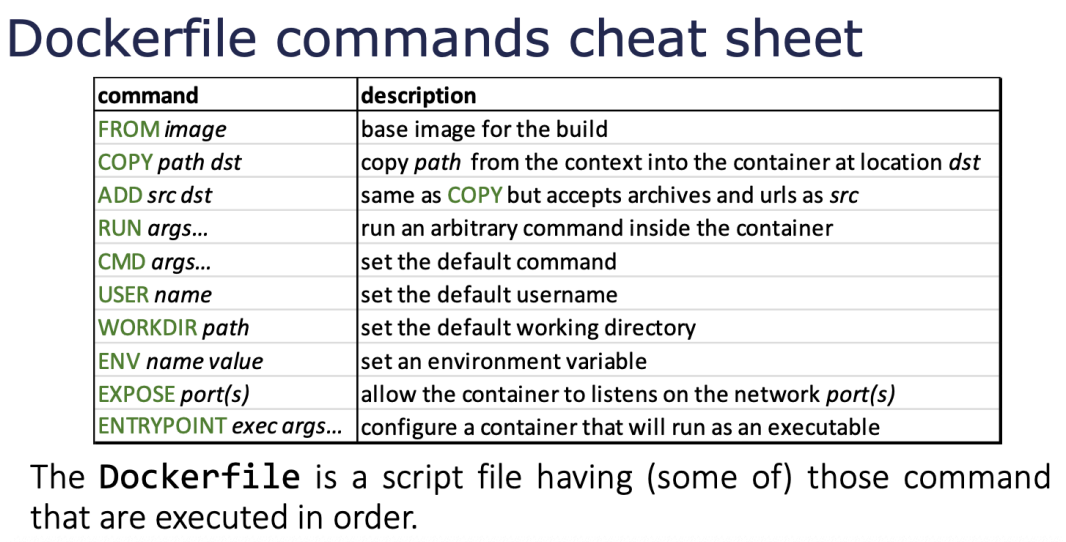
- A container is the running copy.
- The image is the template used to make it.

Images are usually built from a file called a **Dockerfile**.

---

3. Dockerfile: is a simple text file with instructions on how to build an image.

If I change something in the Dockerfile (or in the code copied in the image) I have to build again the image and run the container again.



Dockerfile commands cheat sheet

command	description
FROM *image*	base image for the build
COPY *path dst*	copy *path* from the context into the container at location *dst*
ADD *src dst*	same as COPY but accepts archives and urls as *src*
RUN *args...*	run an arbitrary command inside the container
CMD *args...*	set the default command
USER *name*	set the default username
WORKDIR *path*	set the default working directory
ENV *name value*	set an environment variable
EXPOSE *port(s)*	allow the container to listens on the network *port(s)*
ENTRYPOINT *exec args...*	configure a container that will run as an executable

The `Dockerfile` is a script file having (some of) those command that are executed in order.

---

4. Docker Engine: This is the main software running in the background that:

- Creates and runs containers
- Builds images
- Manages networks and storage

---

5. Registry: is a place where Docker images are stored and shared.

Most common:

- Docker Hub (public)
- Private registries for companies

---

6. Volume: is storage that containers can use to save data permanently.

If the container is deleted, the volume remains. Example: Database files.

Dockerflie Code:

```
FROM python:3.12-slim

WORKDIR /app

COPY requirements.txt .

RUN pip install -r requirements.txt

COPY app.py .

EXPOSE 5000

ENV FLASK_APP=app.py
CMD ["flask", "run", "--host=0.0.0.0"]
```

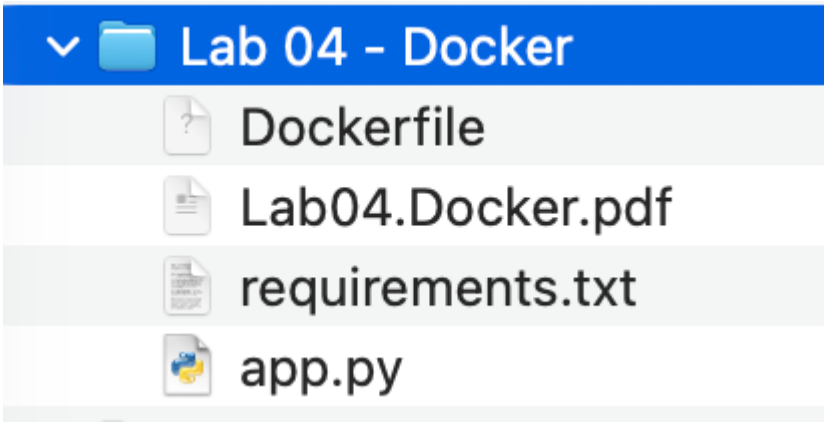
Line	Explanation
`FROM python:3.12-slim`	This tells Docker which **base image** to start from. Here it uses an official **Python 3.12 slim version**— a lightweight Linux image with Python preinstalled.
`WORKDIR /app`	Sets the **working directory** inside the container to `/app`. Every command after this runs inside `/app`. If `/app` doesn't exist, Docker creates it automatically.
`COPY requirements.txt .`	Copies the `requirements.txt` file from your **host machine** (your PC) to the **current directory inside the container** (here `/app`). ➡️ The dot `.` means "copy to **current directory**".
`RUN pip install -r requirements.txt`	Executes a command **inside the container** during build time. It installs all Python packages listed in `requirements.txt` using pip.

Line	Explanation
`COPY app.py .`	Copies your `app.py` file from your computer into the container's current directory `/app`. Again, `.` means "put it here (in /app)".
`EXPOSE 5000`	Declares that the app **listens on port 5000** inside the container. This doesn't actually open the port — it just documents and signals which port should be mapped when you run it.
`ENV FLASK_APP=app.py`	Sets an **environment variable** inside the container. Flask uses `FLASK_APP` to know which file to run when you start the server.
`CMD ["flask", "run", "--host=0.0.0.0"]`	The command Docker runs when the container starts. `flask run` starts the Flask development server, and `--host=0.0.0.0` makes it listen on **all network interfaces** (so it's accessible from outside the container).

Lab:

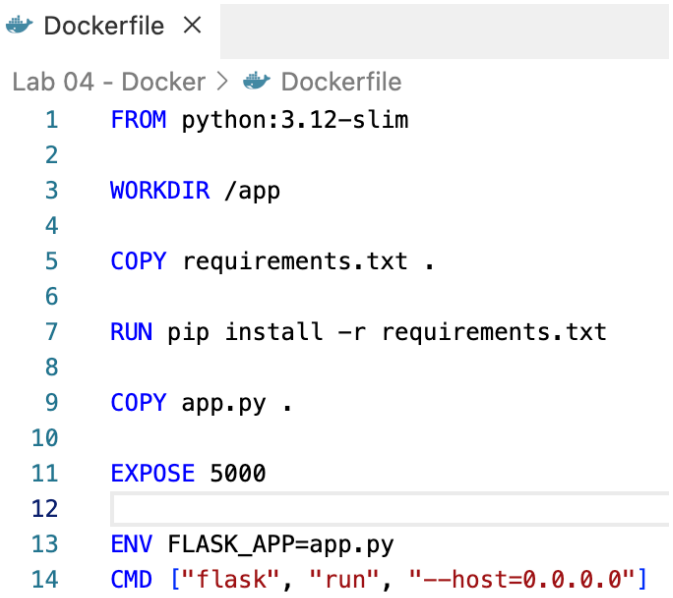
Step 0: Set up Project, by installing `app.py` from the lab-1 and `requirements.txt` to the project folder.

Then I opened this Project folder in VS Code.



Step 1: Writing and creating **Dockerfile**

As, described in VS Code, I created Dockerfile. Then started to fill it up with basic code to create Docker Image. Dockerfile:



Code:

```
FROM python:3.12-slim

WORKDIR /app

COPY requirements.txt .

RUN pip install -r requirements.txt

COPY app.py .

EXPOSE 5000

ENV FLASK_APP=app.py
CMD ["flask", "run", "--host=0.0.0.0"]
```

Line	Explanation
`FROM python:3.12-slim`	This tells Docker which **base image** to start from. Here it uses an official **Python 3.12 slim version**— a lightweight Linux image with Python preinstalled.
`WORKDIR /app`	Sets the **working directory** inside the container to `/app`. Every command after this runs inside `/app`. If `/app` doesn't exist, Docker creates it automatically.

Line	Explanation
`COPY requirements.txt .`	Copies the `requirements.txt` file from your **host machine** (your PC) to the **current directory inside the container** (here `/app`). ➡️ The dot `.` means “copy to **current directory**”.
`RUN pip install -r requirements.txt`	Executes a command **inside the container** during build time. It installs all Python packages listed in `requirements.txt` using pip.
`COPY app.py .`	Copies your `app.py` file from your computer into the container’s current directory `/app`. Again, `.` means “put it here (in /app)”.
`EXPOSE 5000`	Declares that the app **listens on port 5000** inside the container. This doesn’t actually open the port — it just documents and signals which port should be mapped when you run it.
`ENV FLASK_APP=app.py`	Sets an **environment variable** inside the container. Flask uses `FLASK_APP` to know which file to run when you start the server.
`CMD ["flask", "run", "--host=0.0.0.0"]`	The command Docker runs when the container starts. `flask run` starts the Flask development server, and `--host=0.0.0.0` makes it listen on **all network interfaces** (so it’s accessible from outside the container).

`COPY requirements.txt requirements.txt` is the same but the one we used is more clean

Step 2.0: Before docker build, we should know that our docker application should be running, to create docker image at this point. So we open Docker application and can continue. Remember that our `python:3.12-slim` where installed before we started lab work, so it was settled inside docker app well.

Step 2: Building **Docker Image** using `docker build` command.

Open the terminal, route should be inside the project folder, and run this command:

`docker build -t my-flask-app .`

- `docker build` is the command to build an image.
- `t mylab4` **tags** (names) your image "mylab4"
- `.` tells Docker to look for the Dockerfile in the current directory, **it should have space before itself.**

Step 3: We Run two containers. The lab requires you to run two containers from the same image, each on a different port. You will need **two separate terminals** for this.

In Terminal 1: Run the first container, mapping your machine's port 5001 to the container's port 5000. `docker run --name app1 -p 5001:5000 mylab4`

In Terminal 2: Run the second container, mapping your machine's port 5002 to the container's port 5000. `docker run --name app2 -p 5002:5000 mylab4`

Command Breakdown:

- `docker run` creates and starts a new container.
- `-name app1` gives your container a custom name for easy reference.
- `p 5001:5000` is the **port mapping**. It links the `<host_port>` (5001) to the `<container_port>` (5000).
- `mylab4` is the name of the image you want to run.

Step 4: We Test our Containers.

- **To test the first container:** Open your browser and go to `http://127.0.0.1:5001/hello`
- **To test the second container:** Open your browser and go to `http://127.0.0.1:5002/hello`

In both cases, you should see the JSON response from your application: `{"message": "Hello world!"}`

The Role of the Path (`/hello`)

The part after the port (`/hello`) tells the **Flask application** *inside* the container what specific function or logic you want to execute. If you look at your `app.py` code, you will see the definition:

Python:

```
@app.route('/hello', methods=['GET'])
def hello():
    return jsonify(message="Hello world!")
```

Step 5: Clean Up

We do to each terminal and stop each container with `control+c`

We remove containers to clean up space: `docker rm app1` & `docker rm app2`

`docker rm` **- removes container**

> **`docker ps -a`: Проверяет все запущенные контейнеры на нашем устройстве, важно их удалять и проверять.**

✅ Lab 5: Microservices and Docker Compose

Theory:

Docker Compose

A tool to run multiple containers at the same time (e.g., a web server + database).

Configuration is written in a `docker-compose.yml` file.

`docker-compose.yml` example:

```
services:
  calc:
    build: ./calc
    container_name: calc
    ports:
      - "5001:5001"
    depends_on:
      - last

  string:
    build: ./string
    container_name: string
    ports:
      - "5003:5003"
    depends_on:
      - last

  last:
    build: ./last
    container_name: last
```

```
    ports:
      - "5002:5002"
    volumes:
      - lastvolume:/app
volumes:
  lastvolume:
```

Code Understanding:

`services:` This section defines **all containers** that Docker Compose will create and run.

`build: ./calc` Docker looks in the ./calc directory → It finds the Dockerfile there → It builds an image based on that Dockerfile.

`container_name: calc` The running container will be named calc.

`ports: - "5001:5001"` This exposes the container's internal port 5001 to your computer's port 5001: left side 5001 → your browser / host machine. right side 5001 → Flask app inside container

`depends_on: - last` This means: last service must start first, but does NOT wait for it to become ready

`volumes: - lastvolume:/app` This is VERY IMPORTANT:

- `lastvolume` = Here is Docker persistent volume
- `/app` = folder inside the last-service container

Meaning: The `last.txt` file created by last-service will survive container restarts. Other services cannot access it directly — only through HTTP. You can rebuild containers, but last.txt stays

`volumes: lastvolume:` "Create a named persistent volume called `lastvolume`." It will be stored on your system (not inside the container).

Also there could be `restart: always` inside service. Which simply means if there is a failure, restart always.

About the Docker Compose Network:

In Dockerfiles and docker-compose files:

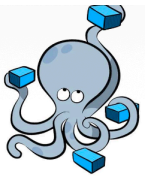
- Expose the ports that other containers in the docker-compose network should be able to use.

- If you want a microservice to be reachable from *outside* docker-compose (for example, from your browser), you must map its ports to the host machine.

Inside your code (running inside the docker-compose network):

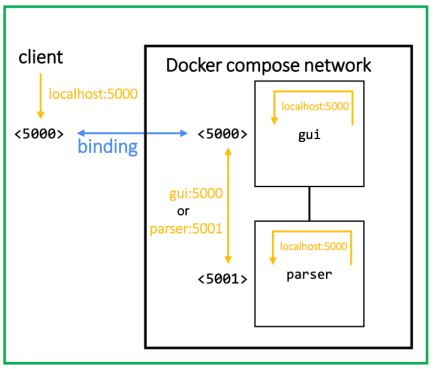
- Make each service listen on the port you exposed.
- When one service needs to call another, use the other service's name and the port it exposes.
- Using `localhost` inside a container refers only to *that* container, not your computer.
- If you need to reach something outside the docker-compose network, it is possible, but the method depends on your operating system. You can look up the correct approach online.

About Docker compose network



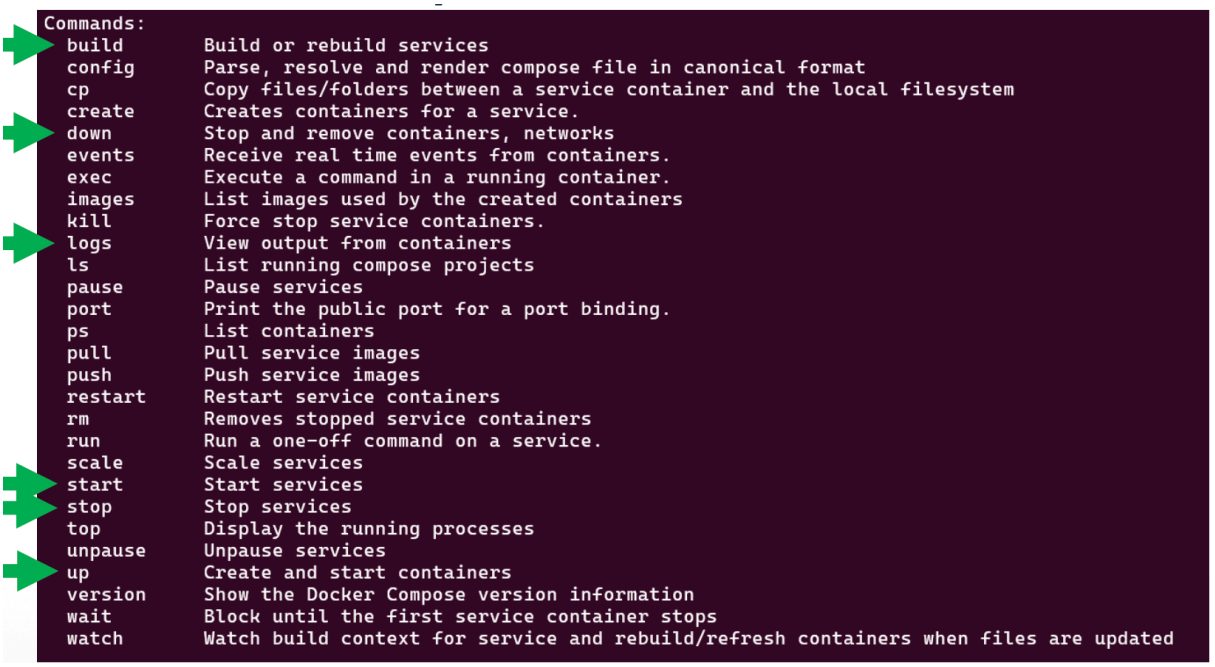
- **`parser`** and **`gui`** can communicate using their exposed port (5000 and 5001) and their name, e.g. http://parser:5001
- **`parser's`** port is bound with the 5000 port of the machine, using **`ports`** with **`5000:5000`**. It is equivalent to running a container with **`-p 5000:5000`**.
- From outside the docker compose network, ports 5000 and 5001 of the microservices are not reachable directly. A client can send requests only to port 5000 of the machine, and the request will be forwarded to the 5000 of **`gui`** because the ports are bound.
- From the code of **`gui`** and **`parser`**, using **`127.0.0.1`** (localhost) means the localhost of the container, not the localhost of the local machine.
- Note that the ports are unique per machine or per container. This means that the **`parser`**'s exposed port can be 5000 without problems, and the machine port can be a different one as well. What you cannot do is bind the same port of your local machine multiple times.

Local machine network



`gui` Dockerfile exposes 5000
`parser` Dockerfile exposes 5001

Docker Compose Commands:



Lab:

You take your old monolithic Flask app from Lab 4 and **split it into 3 microservices**, then run them using **Docker Compose**.

We will create 3 microservices:

1. **calc** → math operations
2. **string** → string operations
3. **last** → storage of last operation (with /last and /notify)

Previously in the Lab 4 we had monolith system, which runs every operation, service in the one single **dockerfile, (docker container)**. And all of those services was in one `app.py` like this:

`add, sub, mul, div, mod, random, upper, lower, concat, reduce, crash, last`

Code of Monolith Lab 4:

```
from flask import Flask, request, make_response, jsonify
import random, time, os, threading

app = Flask(__name__)
```

```
@app.route('/add')
def add():
    a = request.args.get('a', type=float)
    b = request.args.get('b', type=float)
    if a and b:
        save_last("add",(a,b),a+b)
        return make_response(jsonify(s=a+b), 200) #HTTP 200 OK
    else:
        return make_response('Invalid input\n', 400) #HTTP 400 BAD REQUES
T

@app.route('/sub')
def sub():
    a = request.args.get('a', type=float)
    b = request.args.get('b', type=float)
    if a and b:
        save_last("sub",(a,b),a-b)
        return make_response(jsonify(s=a-b), 200)

@app.route('/mul')
def mul():
    a = request.args.get('a', type=float)
    b = request.args.get('b', type=float)
    if a and b:
        save_last("mul",(a,b),a*b)
        return make_response(jsonify(s=a*b), 200)
    else:
        return make_response('Invalid input\n', 400)

@app.route('/div')
def div():
    a = request.args.get('a', type=float)
    b = request.args.get('b', type=float)
    if a and b:
        if b == 0:
            return make_response('Division by zero\n', 400)
        save_last("div",(a,b),a/b)
        return make_response(jsonify(s=a/b), 200)
```

```
    else:
        return make_response('Invalid input\n', 400)

@app.route('/mod')
def mod():
    a = request.args.get('a', type=float)
    b = request.args.get('b', type=float)
    if a and b:
        if b == 0:
            return make_response('Division by zero\n', 400)
        save_last("mod",(a,b),a%b)
        return make_response(jsonify(s=a%b), 200)
    else:
        return make_response('Invalid input\n', 400)

@app.route('/random')
def rand():
    a = request.args.get('a', type=int)
    b = request.args.get('b', type=int)
    if a and b:
        if a > b:
            return make_response('Invalid input\n', 400)
        res = random.randint(a, b)
        save_last("random",(a,b),res)
        return make_response(jsonify(s=res), 200)
    else:
        return make_response('Invalid input\n', 400)

@app.route('/reduce')
def reduce():
    op = request.args.get('op', type=str)
    lst = request.args.get('lst', type=str)
    if op and lst:
        lst = eval(lst)
        if op == 'add':
            res = sum(lst)
            response =  make_response(jsonify(s=res), 200)
        elif op == 'sub':
```

```
            res = lst[0] - sum(lst[1:])
            response = make_response(jsonify(s=res), 200)
        elif op == 'mul':
            res = 1
            for i in lst:
                res *= i
            response = make_response(jsonify(s=res), 200)
        elif op == 'div':
            res = lst[0]
            for i in lst[1:]:
                if i == 0:
                    return make_response('Division by zero\n', 400)
                res /= i
            response = make_response(jsonify(s=res), 200)
        elif op == 'concat':
            res = ""
            for i in lst:
                res += i
            response = make_response(jsonify(s=res), 200)
        else:
            return make_response(f'Invalid operator: {op}', 400)
        save_last("reduce",(op,lst),res)
        return response
    else:
        return make_response('Invalid operator\n', 400)

@app.route('/concat')
def concat():
    a = request.args.get('a', type=str)
    b = request.args.get('b', type=str)
    if a and b:
        res = a+b
        save_last("concat",(a,b),res)
        return make_response(jsonify(s=res), 200)
    else:
        return make_response('Invalid input\n', 400)

@app.route('/upper')
```

```
def upper():
    a = request.args.get('a', 0, type=str)
    res = a.upper()
    save_last("upper","("+a+")",res)
    return make_response(jsonify(s=res), 200)

@app.route('/lower')
def lower():
    a = request.args.get('a', 0, type=str)
    res = a.lower()
    save_last("lower","("+a+")",res)
    return make_response(jsonify(s=res), 200)

@app.route('/crash')
def crash():
    def close():
        time.sleep(1)
        os._exit(0)
    thread = threading.Thread(target=close)
    thread.start()
    ret = str(request.host) + " crashed"
    return make_response(jsonify(s=ret), 200)

@app.route('/last')
def last():
    try:
        with open('last.txt', 'r') as f:
            return make_response(jsonify(s=f.read()), 200)
    except FileNotFoundError:
        return make_response('No operations yet\n', 404)


def save_last(op,args,res):
    with open('last.txt', 'w') as f:
            f.write(f'{op}{args}={res}')

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

Step 1: We need to split the monolith to microservices in that order.

Microservice - 1. Calculation operations:

- `/add, /sub, /mul, /div, /mod, /random, /reduce`

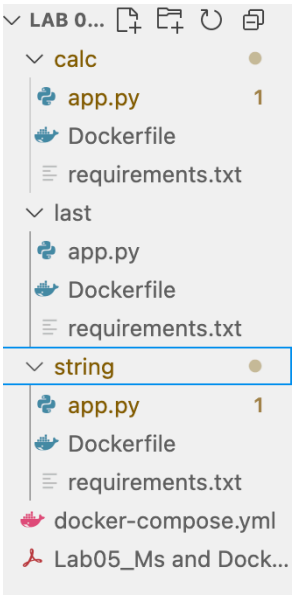
Microservice - 2. String operations:

- `/upper, /lower, /concat`

Microservice - 3. Last operation: (Service that shows last operation from above 2 microservices)

- `/last` returns last operation from a file or memory
- `/notify` accepts POST JSON with format:

Project Structure:



Step 2: Then we have to change the code of `app.py` for every microservice splitting it into 3 different.

1. `calc\app.py:`

```
from flask import Flask, request, jsonify, make_response
import requests
import random
```

```
app = Flask(__name__)

LAST_SERVICE_URL = "http://last:5002/notify"

def notify_last(op, args, res):
    try:
        requests.post(LAST_SERVICE_URL, json={
            "op": f"{op}{args}",
            "result": res
        })
    except:
        pass


@app.route('/add')
def add():
    a = request.args.get('a', type=float)
    b = request.args.get('b', type=float)
    if a is None or b is None:
        return make_response("Invalid input\n", 400)

    res = a + b
    notify_last("add", (a, b), res)
    return jsonify(s=res)


@app.route('/sub')
def sub():
    a = request.args.get('a', type=float)
    b = request.args.get('b', type=float)
    if a is None or b is None:
        return make_response("Invalid input\n", 400)

    res = a - b
    notify_last("sub", (a, b), res)
    return jsonify(s=res)
```

```
@app.route('/mul')
def mul():
    a = request.args.get('a', type=float)
    b = request.args.get('b', type=float)
    if a is None or b is None:
        return make_response("Invalid input\n", 400)

    res = a * b
    notify_last("mul", (a, b), res)
    return jsonify(s=res)


@app.route('/div')
def div():
    a = request.args.get('a', type=float)
    b = request.args.get('b', type=float)

    if a is None or b is None:
        return make_response("Invalid input\n", 400)
    if b == 0:
        return make_response("Division by zero\n", 400)

    res = a / b
    notify_last("div", (a, b), res)
    return jsonify(s=res)


@app.route('/mod')
def mod():
    a = request.args.get('a', type=float)
    b = request.args.get('b', type=float)

    if a is None or b is None:
        return make_response("Invalid input\n", 400)
    if b == 0:
        return make_response("Division by zero\n", 400)

    res = a % b
```

```
    notify_last("mod", (a, b), res)
    return jsonify(s=res)


@app.route('/random')
def rand():
    a = request.args.get('a', type=int)
    b = request.args.get('b', type=int)

    if a is None or b is None or a > b:
        return make_response("Invalid input\n", 400)

    res = random.randint(a, b)
    notify_last("random", (a, b), res)
    return jsonify(s=res)


@app.route('/reduce')
def reduce_route():
    op = request.args.get('op', type=str)
    lst = request.args.get('lst')

    if not op or not lst:
        return make_response("Invalid input\n", 400)

    lst = eval(lst)

    if op == "add":
        res = sum(lst)
    elif op == "sub":
        res = lst[0] - sum(lst[1:])
    elif op == "mul":
        res = 1
        for i in lst:
            res *= i
    elif op == "div":
        res = lst[0]
        for i in lst[1:]:
```

```
            if i == 0:
                return make_response("Division by zero\n", 400)
            res /= i
    elif op == "concat":
        res = "".join(lst)
    else:
        return make_response("Invalid operator\n", 400)

    notify_last("reduce", (op, lst), res)
    return jsonify(s=res)


if __name__ == '__main__':
    app.run(host="0.0.0.0", port=5001)
```

2. string\app.py:

```
from flask import Flask, request, jsonify, make_response
import requests

app = Flask(__name__)

LAST_SERVICE_URL = "http://last:5002/notify"

def notify_last(op, args, res):
    try:
        requests.post(LAST_SERVICE_URL, json={
            "op": f"{op}{args}",
            "result": res
        })
    except:
        pass


@app.route('/upper')
def upper():
    s = request.args.get('s')
```

```
    if not s:
        return make_response("Invalid input\n", 400)

    res = s.upper()
    notify_last("upper", (s,), res)
    return jsonify(s=res)


@app.route('/lower')
def lower():
    s = request.args.get('s')
    if not s:
        return make_response("Invalid input\n", 400)

    res = s.lower()
    notify_last("lower", (s,), res)
    return jsonify(s=res)


@app.route('/concat')
def concat():
    a = request.args.get('a')
    b = request.args.get('b')
    if a is None or b is None:
        return make_response("Invalid input\n", 400)

    res = a + b
    notify_last("concat", (a, b), res)
    return jsonify(s=res)


if __name__ == '__main__':
    app.run(host="0.0.0.0", port=5003)
```

3. last\app.py:

```
from flask import Flask, request, jsonify, make_response
import json
import os

app = Flask(__name__)

LAST_FILE = "last.txt"


@app.route('/notify', methods=['POST'])
def notify():
    data = request.get_json()

    if not data or "op" not in data or "result" not in data:
        return make_response("Invalid notify payload\n", 400)

    with open(LAST_FILE, "w") as f:
        f.write(json.dumps(data))

    return make_response("OK\n", 200)


@app.route('/last', methods=['GET'])
def last():
    if not os.path.exists(LAST_FILE):
        return make_response("No operations yet\n", 404)

    with open(LAST_FILE, "r") as f:
        content = json.loads(f.read())

    return jsonify(content)


if __name__ == '__main__':
    app.run(host="0.0.0.0", port=5002)
```

Differences of code in monolith `app.py` and 3 microservices:

1. **Saving “last” result**

Monolith: saves directly to a file:

```
def save_last(op, args, res):
    with open('last.txt', 'w') as f:
        f.write(f'{op}{args}={res}')
```

Microservices: replaced with **HTTP call** to another container:

```
def notify_last(op, args, res):
    requests.post("http://last:5002/notify", json={
        "op": f"{op}{args}",
        "result": res
    })
```

📌 **Biggest difference**:

Monolith writes to a file → microservices send the data over the network.

---

2. Reading the last result

Monolith: Reads local file.

```
@app.route('/last')
def last():
    with open('last.txt', 'r') as f:
        return jsonify(s=f.read())
```

Microservice #3 (Last Service): A **dedicated service** does the reading:

```
@app.route('/last')
def last():
    with open("last.txt", "r") as f:
        return jsonify(json.loads(f.read()))
```

📌 Only the **last-service** handles the file.

---

3. Flask run differences

Monolith

```
app.run(debug=True)
```

Microservices: Each explicitly sets host + port for Docker:

```
app.run(host="0.0.0.0", port=5001)   # math
app.run(host="0.0.0.0", port=5003)   # string
app.run(host="0.0.0.0", port=5002)   # last
```

📌 Needed so containers can talk over the Docker network.

---

4. File format changed

Monolith: Writes plain text: `add(5,3)=8`

Microservices: Writes JSON: `{"op": "add(5,3)", "result": 8}`

Step 3: Changing the `requirements.txt` for every service. For (Calc and String requirements.txt) It should be:
`Flask`

`requests`

and for Last requirements.txt: `Flask`

Step 4: Creating **Dockerfile** for each Microservice, they will be the same except their **Expose ports: 5001, 5002, 5003**

```
FROM python:3.12-slim

WORKDIR /app

COPY app.py .
COPY requirements.txt .

RUN pip install -r requirements.txt

EXPOSE 5001 #(5002, 5003 for each microservice)

CMD flask run --host=0.0.0.0 --port=5001 #(5002, 5003)
```

Step 5: We create **Docker Compose** file `(docker-compose.yml)` in the root folder of the Project.

```
services:
  calc:
    build: ./calc
    container_name: calc
    ports:
      - "5001:5001"
    depends_on:
      - last

  string:
    build: ./string
    container_name: string
    ports:
      - "5003:5003"
    depends_on:
      - last

  last:
    build: ./last
    container_name: last
    ports:
      - "5002:5002"
    volumes:
      - lastvolume:/app
volumes:
  lastvolume:
```

Code Understanding:

`services:` This section defines **all containers** that Docker Compose will create and run.

`build: ./calc` Docker looks in the ./calc directory → It finds the Dockerfile there → It builds an image based on that Dockerfile.

`container_name: calc` The running container will be named calc.

`ports: - "5001:5001"` This exposes the container's internal port 5001 to your computer's port 5001: left side 5001 → your browser / host machine. right side 5001 → Flask app inside container

`depends_on: - last` This means: last service must start first but does NOT wait for it to become ready

`volumes: - lastvolume:/app` This is VERY IMPORTANT:

lastvolume = Docker persistent volume

/app = folder inside the last-service container

Meaning: The `last.txt` file created by last-service will survive container restarts. Other services cannot access it directly — only through HTTP. You can rebuild containers, but last.txt stays

`volumes: lastvolume:` "Create a named persistent volume called `lastvolume`." It will be stored on your system (not inside the container).

Step 6: Run the Microservice. In the root folder where `docker-compose.yml` file is located. **First Build:**

`docker compose up --build`

Its used for the first time to build actual containers, for the next time we can run it just using: `docker compose up` saving time.

Then when we see that 3 of our microservices running in parallel in 3 different containers at the same time, we can check if they are really working.

Step 7: Checking the Result.

Test calc: http://localhost:5001/add?a=5&b=7

Test string: http://localhost:5003/upper?s=hello

Test last: http://localhost:5002/last

Step 8: Close the file, delete the containers.

Important!

> After finishing my Lab or Work with ASE It is important to delete all the **containers, images and volumes** that was created.

Guidline to delete, clean the space from everything related to docker:

https://www.digitalocean.com/community/tutorials/how-to-remove-docker-images-containers-and-volumes#purging-all-unused-or-dangling-images-containers-volumes-and-networks

- `docker system prune` **deletes everything**
- `docker system prune -a` also deletes everything.
- `docker volume prune` to remove all unused volumes

- Containers: `docker ps -a` → should show nothing
- Images: `docker images` → should show nothing
- Volumes: `docker volume ls` → should show nothing
- Networks: `docker network ls` → should only show default networks

✅ Lab 6: Testing

Theory:

Testing Microservices

- **Functional testing:** Check whether the whole system works correctly. This usually happens step-by-step:

  Unit Test → Feature Test → System Test → Release Testing.

- **User testing:** Real users try the application to see if it is easy to use and meets their needs.
- **Performance testing:** Measure how fast each microservice works and how it behaves when the workload increases.
- **Security testing:** Check the system for weaknesses or possible attacks.

Unit testing is a method where you test the smallest pieces of your code one by one to make sure each piece works correctly on its own.

A "unit" can be: single function, class, API endpoint, or even a small microservice. The goal is to catch problems early, before these parts are combined with the rest of the system.

For example: Imagine testing a single function `div (a,b)` .To do the unit test, you need:

- A set of input values for a and b
- The expected correct output of each test case.
- An assertion that compares the execution of the function with the expected output.

```
import pytest

def div(a,b):
    return a / b

def test_div_positive numbers():
    assert div (10, 2) == 5
```

Function to test: `def div(a,b):`

Function call with inputs: `div (10, 2)`

Expected correct result: `== 5`

Assertion of equivalence: `assert div (10, 2) == 5` If assertion is true the test is passed otherwise is failed.

Functional tests in a microservice project call the microservice through its real API. They send HTTP requests to the endpoints and check whether the returned responses are correct.

Why they are important:

- They verify that the service actually does what it is supposed to do.
- They confirm that bugs you previously fixed do not come back.

Test in isolation is done without other microservices. *How can service work and tested if it needs to send logs to the DB Manager ?*

We use mock in testing, a **mock** is a fake version of a component, function, or service used to simulate real behavior without actually performing the real operation.

- **Purpose:** To test a piece of code in isolation without relying on external systems (like databases, APIs, or other services).
- **Example:** If your code calls an API to get user data, you can use a mock to return a fixed response instead of actually calling the API. This makes tests faster, reliable, and predictable.

After testing the microservices in isolation, we test the architecture through the gateway. In this phase “testing code” used to mock dependencies is not needed.

Load Test: The goal of a load test is to understand your service's bottlenecks under stress. Understanding your system limits will help you determining how you want to deploy it and if its design is future-proof in case the load increases.

Mock understanding:

In this example for and HTTP GET. This function is called from other routes. `app.py:`

```
def function (op):
    r = requests. get(f'http://other service:5000/{op})
    return r.json()
```

We have dependenciy on other-service’s JSON. In simple cases, we can only modify the code to have mock behaviour. `app.py`:

```
mock_fun = None
def function (op):
    if mock_fun:
        return mock_fun(op)
    else:
        r = requests. get(f'http://other service:5000/{op})
        return r.json()
```

Then we create a different file which imports the microservice code and implements the mock function. This file will be used to execute isolation tests. `test_app.py`:

```
import app as main_app

flask_app = main_app.ap #Flask app
def mock_fun(op): #Implementation
    if op == 'add':
        return {'s': 5}
    elif op == 'sub':
        return {'s': 3}
#Assigning the implementation in app.py
main_app.mock_fun = mock_fun
```

There are several patterns to seperate the test code. **Mock libraries** allow to configure more sophisticated answers: Random choice from multiple values, Error for negative testing, complex objects, database interaction etc.

After adding mock code, we have to run microservice alone:

- Using separate Dockerfile for testing, or
- Using the “production” Dockerfile but overriding CMD when running the container in order to use mock code

Postman is a software tool that helps developers create, send, and test HTTP requests to APIs. It provides an easy way to check how an API behaves, automate tests, and save request workflows for future use.

You can organize your requests into collections and customize everything, including: HTTP method and URL, Headers, Authorization, Body etc.

Tests in Postman aka JavaScript scripts that use the Postman pm API.

Run single request or a collection of requests to see the results of testing.

Locust is an open-source tool used for load testing. It uses a file called `locustfile.py` where you describe how simulated users should behave.

To use it, you need to:

- Write your tests inside `locustfile.py` .
- Start the microservice using Docker Compose.
- Run in terminal: `locust`

- Open http://localhost:8089 in your browser, configure the test settings, and start the load test.

Example:

```
from locust import HttpUser, task, between

class QuickstartUser (HttpUser):
    @task()
    def add(self):
        for a in range(1,11):
            self.client.get(f"/calc/add?a={a}&b=9", name = "Calc Add Endpoin
t")
```

This is an example to test the add endpoint 10 times for each user with a different parameter.

- A valid `locustfile` has a class that extends User class (ex: HttpUser).
- Each test is annotated with `@task()` representing a simulated user performs that task

Lab:

For todays lab we had to do different kind of testings. Downloaded **Postman** to do Testing.

1. **Unit, Isolated testings** for each microservice `Calc` and `String`, using Postman.

For this testing we kind of changed `Calc` and `String` `app.py` code to use also, `mock_fun` for testing. And we created `test_app.py` for exactly testing using mock and isolated.

This is example of `test_app.py` which was created for `Calc` operation, also was created for `String` folder.

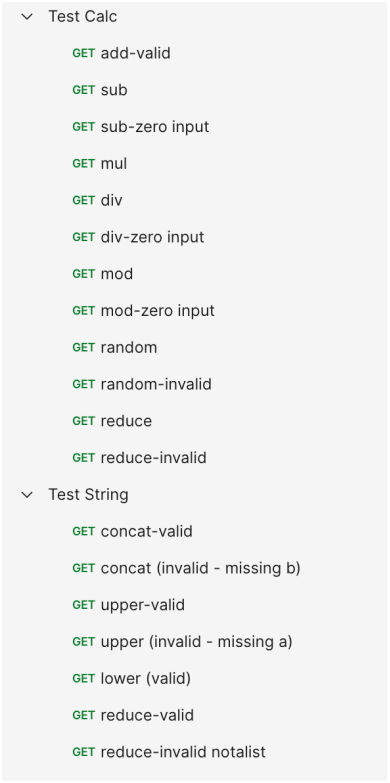
```
import app as main_app
from flask import Flask

# Example mock: simply collects payloads for inspection.
```

```
collected = []
def mock_fun(payload):
    collected.append(payload)
    # could return a value if test expects it
    return {"ok": True, "mocked": True}

# attach the mock so notify_db() will use it
main_app.mock_fun = mock_fun

app = main_app.app

if __name__ == '__main__':
    # Run on port 5001 for local testing
    app.run(host='0.0.0.0', port=5001, debug=True)
```

This means that simply we use, mock data instead of using Database and forwarding it to database, instead we get answer from mock, running it for only testing at the first stage.

Then we changed common problems inside the code which was written with bugs, with hands (**app.py of each two services**).

As the next step we created, wrote **Unit test** in Postman for each microservice separately.



For each Microservice we created **Collection** of tests, and of course for each microservice we used different **port hosts**. like **5001 for Calc** and **5002 for String**.

Then we had two options of testing for Unit Tests:

1. Using and opening inside the each microservice folder, **venv virtual environment**, and running `test_app.py` of each microservice, to make test safe and reliable. When we run inside the `venv`, each microservice. While running each microservice, also we Run **Postman** endpoint tests. Here we can see, fails of code, find bugs using this testing, also changed code after fails and bugs. Then on the last Test we got every test passed for each microservice.
2. **Another option**, which one was to run unit tests for each microservice using **Dockerfiles**, instead of `venv`, other steps are almost the same, and to run exactly `test_app.py` we use:

`docker run string python test_app.py` And other things still remain same, after running our services separately, we test them using **Postman collections for each Service separately**.

Next step is **Integration testing** using `Gateway`. For this testing we use **Docker compose** file, in the root folder of Lab.

We also start from fixing the code of `app.py` for `Gateway` service. Then in **Postman** we create Tests which is kind of collecting all of the previous tests into one single Collection. But using different URL, since Gateway forwards this requests to following 2 microservices.

Here `Gateway` works as a Integration for all the microservices. Here we use `app.py` instead of creating `test_app.py` for `Gateway`.
We create **docker containers** and run docker compose using:

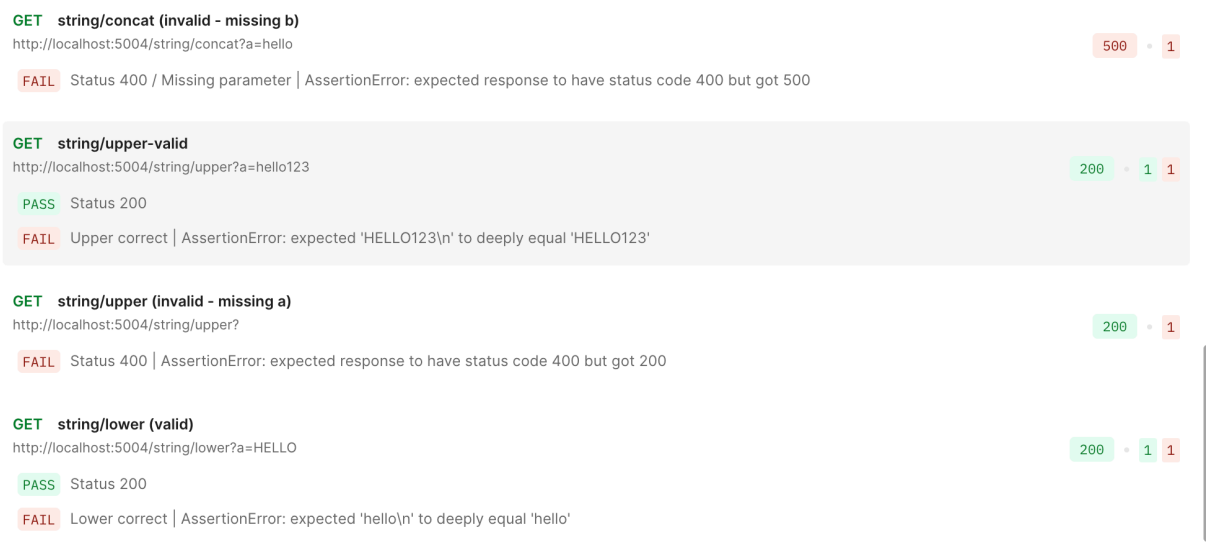
`docker compose up —build` (after finish we run `docker compose down`)

While we running our **Docker Compose**, we test it using new **Collection for Gateway**, the difference here as we said we use `app.py` instead of creating `test_app.py`, and we use different routes endpoints for operations which goes through Gateway:

`http://localhost:5004/calc/add?a=5&b=3`

in each microservice Unit Testing this URL was:
`http://localhost:5001/add?a=5&b=3` because simply, in that case we was running exactly inside the calc folder.

After Integration test, we again, fix code, fix bugs, then pass all the Tests.



Last step, of testing in todays lab is **Performace Test using Locust**.

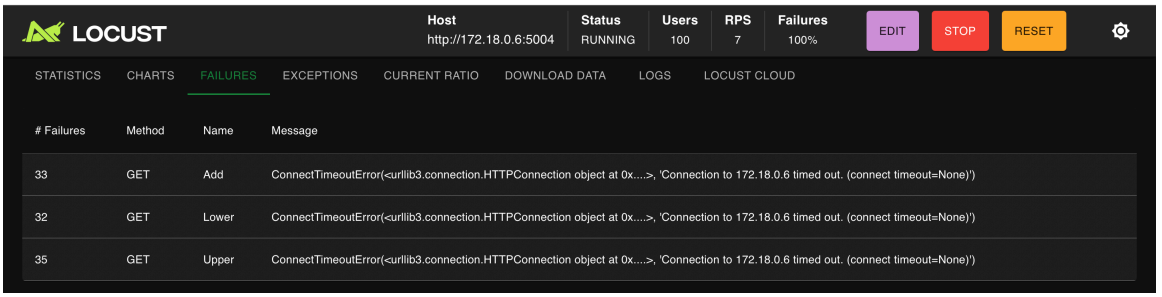
For this in the root folder, we change and add test cases for `locustfile.py`:

```
from locust import HttpUser, task, between

class GatewayLoadTest(HttpUser):

    # Wait time between tasks (simulates real users)
    wait_time = between(0.1, 0.5)

    @task
    def test_add(self):
        self.client.get("/calc/add?a=5&b=3", name="Add")

    @task
    def test_sub(self):
        self.client.get("/calc/sub?a=10&b=3", name="Sub")

    @task
    def test_mod(self):
        self.client.get("/calc/mod?a=10&b=4", name="Mod")

    @task
    def test_random(self):
        self.client.get("/calc/random?a=1&b=10", name="Random")

    @task
    def test_reduce(self):
        self.client.get("/calc/reduce?op=add&lst=[1,2,3]", name="Reduce Cal
c")

    @task
    def test_upper(self):
        self.client.get("/string/upper?a=hello123", name="Upper")

    @task
    def test_lower(self):
        self.client.get("/string/lower?a=HELLO", name="Lower")

    @task
    def test_concat(self):
```

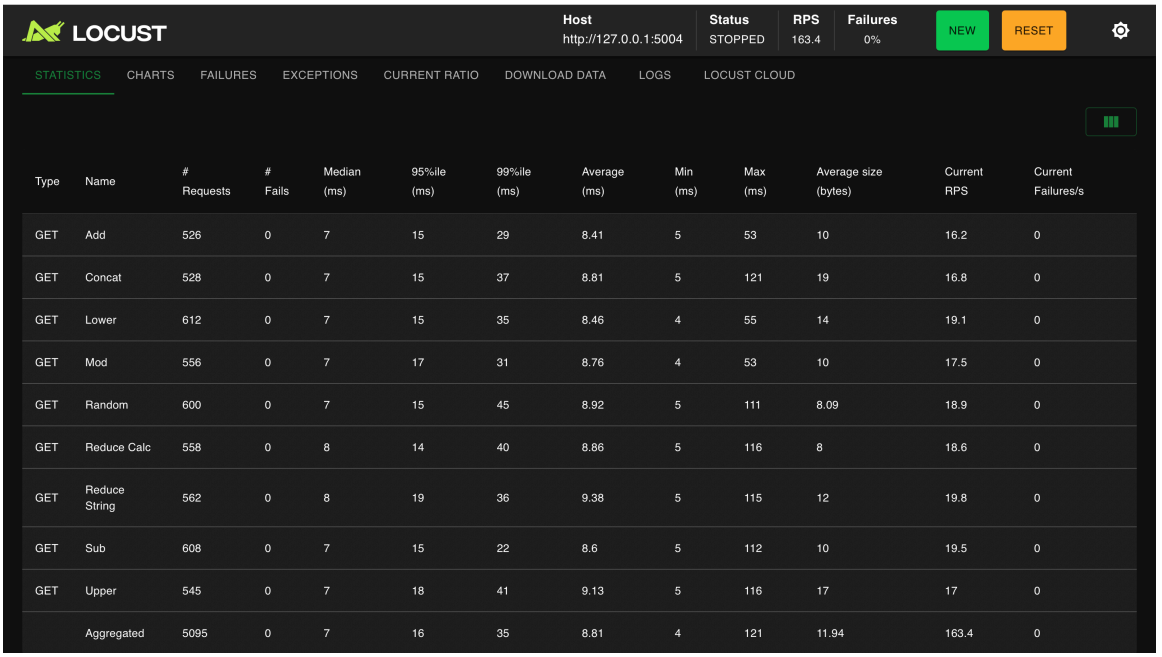
```
        self.client.get("/string/concat?a=hello&b=world", name="Concat")

    @task
    def test_string_reduce(self):
        self.client.get("/string/reduce?op=concat&lst=['a','b','c']",
                        name="Reduce String")
```

This was changed `locustfile.py` with added test cases for locust.

This tests works the same, with running **Docker compose** file. In the different terminal we. write **locust**, and run it in root folder of Lab. Using host: `http://localhost:8089`, we run **locust tests**, this is stress tests, which are tests by possible 1-1000 of users, and how system will behave in such situations, which helps finding bugs, failures.



After we found finished with 0 failures, we can think of it as finished stress test.



✅ Lab 7: DevOps with GitHub Actions

Theory:

CI/CD is a set of practices (way of working) that automate the process of building, testing, and releasing software.

It ensures that code changes are checked early, deployed quickly, and delivered with fewer errors. It helps teams develop faster and more reliably.

Continuous Integration (CI)

- Developers frequently add their code changes to a shared repository, often several times a day.
- Every time they do, automated tools build the project and run tests.
- This helps catch problems early, before they grow into bigger issues.

Continuous Delivery/Deployment (CD)

- **Continuous Delivery:** After the CI process, the system automatically prepares the application for release. It can be deployed to testing or production, but someone must manually approve the final release to production.x
- **Continuous Deployment:** This is fully automatic. If all tests pass, the new version goes straight to production with no manual approval.

CI/CD Pipeline is an automated series of processes triggered by an event that halts if any process fails, ensuring code quality and streamlined delivery.

Github Actions is powerful CI/CD platform that integrates directly with Github repositories. It allows to automate tasks like, testing, building, and deploying your code.

Key Components:

Workflows: Files (written in YAML) that describe what should be automated. They must be stored in the `.github/workflows/` folder in your repository.

Events: Things that start a workflow. Examples: pushing code, opening a pull request, or running on a schedule.

Jobs: A workflow is split into jobs. Each job contains several steps and can run at the same time as others or one after another.

Steps: The individual actions inside a job. Examples: run a script, install packages, build the project, or deploy it.

Runners: The machines that actually perform the jobs. GitHub provides its own runners in the cloud, or you can use your own machine as a self-hosted runner.

GitHub Actions Workflow code:

Event (Trigger Section):

This tells GitHub Actions to start the workflow whenever:

- Someone pushes to `main`
- Someone creates or updates a pull request into `main`

```
name: MicroASE CI
on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main
```

Job Definition: The workflow has one job called **CI-pipeline**. It runs on a GitHub-hosted Ubuntu machine.

```
jobs:
  ci-pipeline:
    runs-on: ubuntu-latest
```

Steps in the Job:

```
steps:
  # 1. Checkout repository: Downloads your repository code into the runner
```

```
so later steps can access it.
  - name: Checkout repository
    uses: actions/checkout@v4

  # 2. Set up Node.js: Installs Node.js version 20. Needed because Newman (Postman CLI) runs on Node.
  - name: Set up Node
    uses: actions/setup-node@v4
    with:
      node-version: '20'

  # 3. Install Newman: Installs Newman globally. Newman runs your Postman test collections from the terminal.
  - name: Install Newman
    run: |
      npm install -g newman

  # 4. Set up Docker Buildx: Enables advanced Docker building features. Needed to build microservice images.
  - name: Set up Docker
    uses: docker/setup-buildx-action@v3

  ############################################################
  # 5. Test CALC microservice
  ############################################################
  - name: Build calc image
    run: docker build -t calc-service ./src/calc-service/

  - name: Run calc container
    run: docker run -d -p 5000:5000 --name calc-container calc-service

       # Executes Postman tests for Calc. If tests fail, the workflow stops (because continue-on-error = false).
  - name: Run calc Newman tests
    run: newman run tests/calc.postman_collection.json
    continue-on-error: false

  - name: Stop calc container
```

```
  if: always()
  run: docker stop calc-container && docker rm calc-container


############################################################
# 6. Test STRING microservice
############################################################
- name: Build string image
  run: docker build -t string-service ./src/string-service/

- name: Run string container
  run: docker run -d -p 5000:5000 --name string-container string-service

- name: Run string Newman tests
  run: newman run tests/string.postman_collection.json
  continue-on-error: false

- name: Stop string container
  if: always()
  run: docker stop string-container && docker rm string-container


############################################################
# 7. Build and run entire architecture
############################################################
# Starts every service defined in docker-compose.yml.
- name: Docker Compose up
  run: docker compose up -d

      # Executes Postman tests that check the system working as a whole.
- name: Run full system tests
  run: newman run tests/full.postman_collection.json
  continue-on-error: false

      # Stops and cleans up all containers.
- name: Stop entire architecture
  if: always()
  run: docker compose down
```

Lab:

This Lab we should do DevOps with Github Actions.

CI/CD Pipeline requirements (detailed steps):

- Checkout the code
- Set up Node
- Install Newman (Postman CLI)
- Set up Docker
- For each microservice (calc, string):
  - Build Docker image
  - Run container detached
  - Execute unit-level Postman tests with Newman
  - Stop container (even if tests fail)
- Build the entire architecture (docker-compose)
- Run architecture detached
- Run the full architecture test with Newman
- Stop containers

Step 1: We open Github Repository and make it Public. There we go to section Actions, and add there workflow actions manually.

It will automatically open `.github/workflow/main.yml` where we will write our workflow based on Instructions given in the Lab.

This is `main.yml` Workflow that we have created based on requirements, and putted into **Git Actions:**

```
name: MicroASE CI

on:
```

```
  push:
    branches:
      - main
  pull_request:
    branches:
      - main

jobs:
  ci-pipeline:
    runs-on: ubuntu-latest

steps:
 # 1. Checkout repository
 - name: Checkout repository
   uses: actions/checkout@v4

 # 2. Set up Node.js
 - name: Set up Node
   uses: actions/setup-node@v4
   with:
     node-version: '20'

 # 3. Install Newman
 - name: Install Newman
   run: |
     npm install -g newman

 # 4. Set up Docker Buildx
 - name: Set up Docker
   uses: docker/setup-buildx-action@v3

 ############################################################
 # 5. Test CALC microservice
 ############################################################
 - name: Build calc image
   run: docker build -t calc-service ./src/calc-service/

 - name: Run calc container
```

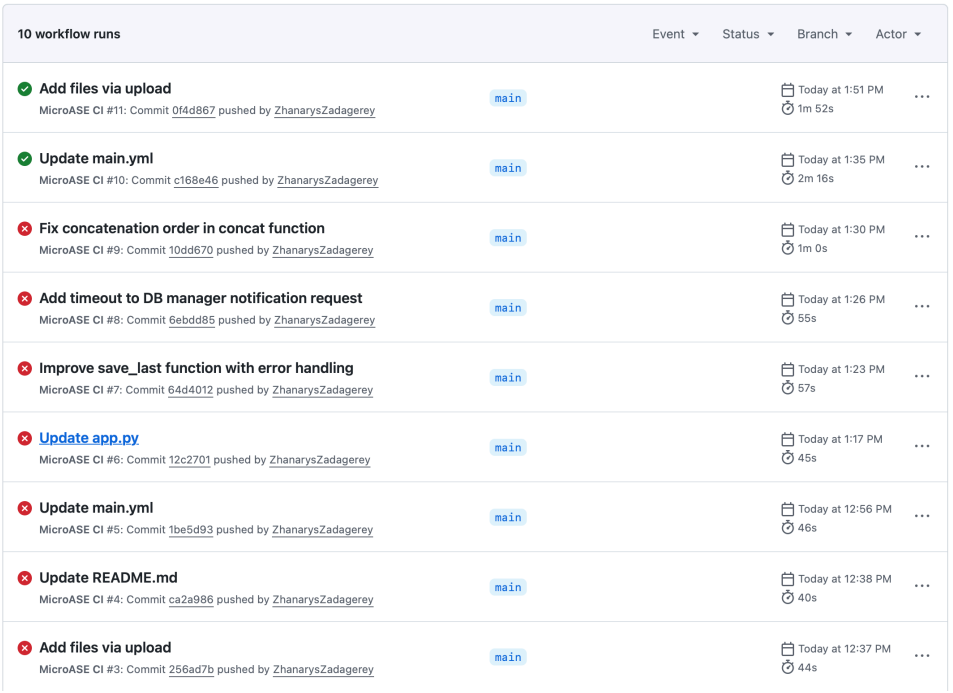
```
  run: docker run -d -p 5000:5000 --name calc-container calc-service

- name: Run calc Newman tests
  run: newman run tests/calc.postman_collection.json
  continue-on-error: false

- name: Stop calc container
  if: always()
  run: docker stop calc-container && docker rm calc-container


############################################################
# 6. Test STRING microservice
############################################################
- name: Build string image
  run: docker build -t string-service ./src/string-service/

- name: Run string container
  run: docker run -d -p 5000:5000 --name string-container string-service

- name: Run string Newman tests
  run: newman run tests/string.postman_collection.json
  continue-on-error: false

- name: Stop string container
  if: always()
  run: docker stop string-container && docker rm string-container


############################################################
# 7. Build and run entire architecture
############################################################
- name: Docker Compose up
  run: docker compose up -d

- name: Run full system tests
  run: newman run tests/full.postman_collection.json
  continue-on-error: false
```

```
- name: Stop entire architecture
  if: always()
  run: docker compose down
```

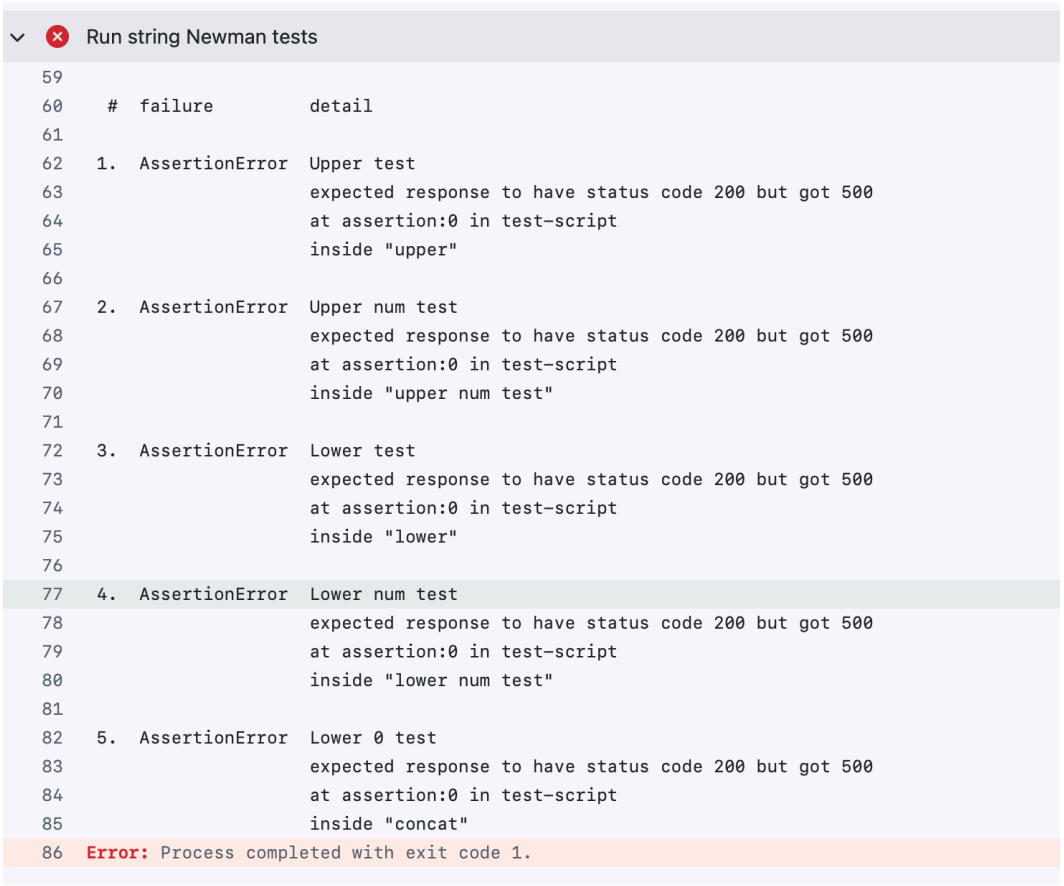
This workflow:

- Runs on every push and PR
- Executes microservice tests first
- Then executes integration tests on full system
- Always shuts down containers, even if tests fail

Step 2: We have pushed it into Github and after Push workflow automatically started to check the code. After this first times we saw, problems in the code, and tests failed, as we can see.



Inside each Workflow runs we can go the Fails and exactly see what was failed detailed.



Then we have fixed all the code problems, bugs. After the Last check and run of Workflow we got Green light, which means we did it right. And with that we finished main part of Lab and we will not go to Bonus stage of Lab.

✅ Lab 8: Data Security

Theory:

Data in transit: Data transmitted from a point to another

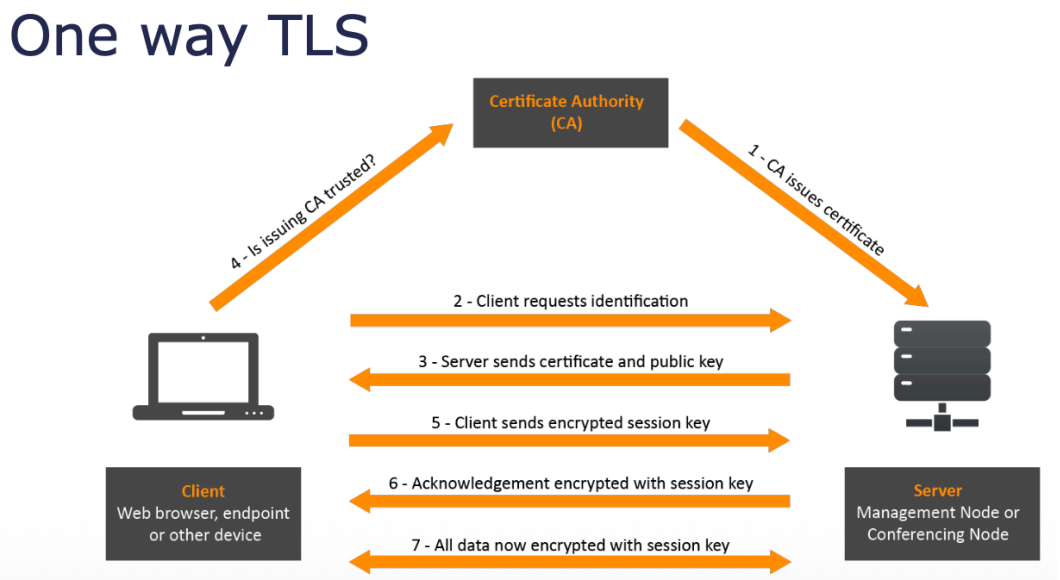
Basic defense: encrypt the communication channel. With services is not practical and safe to use symmetric encryption. To implement asymmetric encryption are used certificates to avoid impersonification.

HTTP + TLS Certificate = HTTPS (TSL = SSL updated version)

One-way TLS:

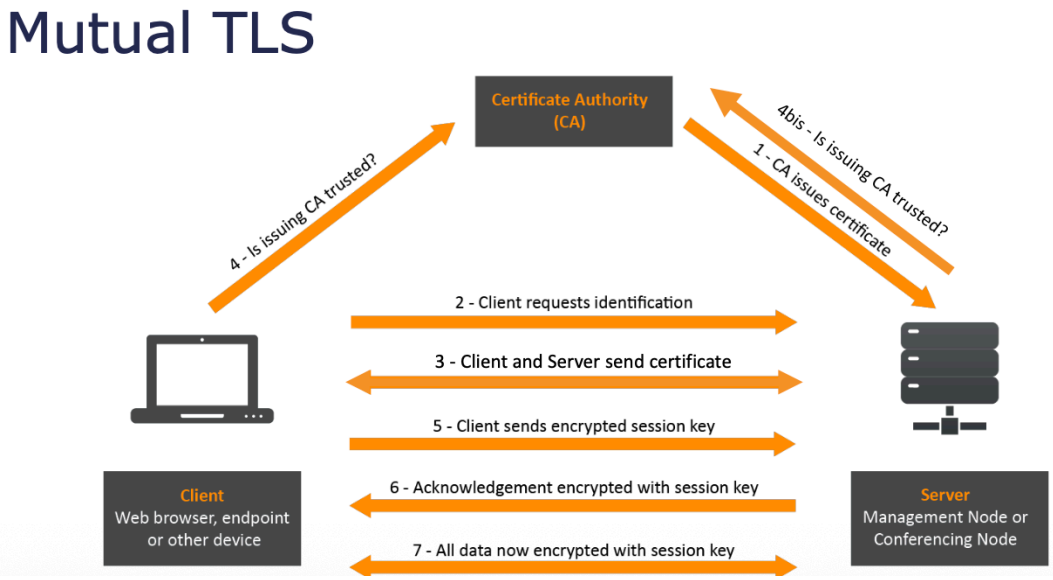
- Client verifies the server's certificate (CA-issued).
- Encrypts the channel; assures server identity only.
- Common for browsers & most public APIs.

- Client identity handled by app layer (tokens/keys/passwords).



Mutual TLS:

- Both client and server present & verify certificates.
- Adds strong client authentication at the transport layer.
- Great for service-to-service, zero-trust, and regulated environments.



A **TLS certificate** contains:

- Public key: used to encrypt and decrypt data.

- Subject: details about the person, organization, or system the certificate is issued to.
- Issuer: the Certificate Authority (CA) that created and signed the certificate.
- Validity period: the dates showing when the certificate starts being valid and when it expires.
- Digital signature: added by the CA to prove that the certificate is genuine and trustworthy.

For a real certificate, you usually need to pay. For development, we can use self-signed certificates, which do not involve a CA. However, they are less secure.

Self Signed Certificates: Use OpenSSL to create a self-signed certificate.

```
openssl req \
  -x509 \ #Generates a x509 certificate
  -newkey rsa:4096 \ #Generates a private key using RSA with 4096bit modulus
  -nodes \ #Avoids encrypting the key (and asking a passphrase)
  -out    cert.pem \ #Name of the output file of the certificate in pem format
  -keyout key.pem \ #Name output file of the key in pem format
  -days 365 \ #Validity period (365 days in this example)
  -subj "/CN=ms" \ #Sets the Subject to Common Name = service
  -addext "subjectAltName=DNS:ms,DNS:localhost,IP:127.0.0.1"
  #Adds Subject Alternative Names for hostname/IP verification
```

This command generates a self-signed certificate and its private key for a service named **ms**, which expires in one year.

Activate HTTPS - Flask:

There are several ways to pass the certificate to a Flask Service, the most common one is to execute it as:

`flask run --cert=cert.pem --key=key.pem ...`

Or use them in the code as:

```
if name == "main":
app.run(ssl_context=('cert.pem','key.pem'))
```

Activate HTTPS - Docker:

Docker compose can manage `secrets` using specific entries in the docker compose file.

```
services:
  myapp:
    image: myapp:latest
    secrets:
      - my_secret
secrets:
  my_secret:
    file: ./my_secret.txt
```

In the `myapp container, /run/secrets/my_secret` is a file set to the contents of the file `./my_secret.txt.`

We need to:

- Create secrets for all certificates and keys (in the docker compose file).
- Pass secrets needed by each microservice (in the docker compose file).
- Allow Flask access certificates and keys (in one of the ways explained before).

Activate HTTPS - Python requests:

We need to change the requests to HTTPS. Example of HTTPS GET:

```
requests.get('https://other:5000/endpoint', verify='run/secrets/other_cert')
```

Our Flask service needs 2 certificates:

- Its own (and its private key).
- The certificate of the service to invoke (other).

The 2 certificates are exchanged and verified during the GET request.

Testing with HTTPS: What is changing?

The protocol changes to HTTPS, and a certificate is exchanged during the connection. With **Postman**, we have to:

- Add **https://** at the beginning of the URL.
- Turn off certificate verification (Settings → General → SSL certificate verification: OFF). **This should only be done in development.**

With a browser, you will see a warning because the certificate is self-signed. You can choose to ignore the warning and continue.

Testing In Isolation with HTTPS:

Also for testing a microservice in isolation we need to use HTTPS. If we need to run container stand-alone with **docker build + run** we can change the testing Dockerfile:

```
FROM python:3.12-slim

WORKDIR /app

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

COPY app*.py .
#Here I copy certificate and key in the same path as docker compose
COPY cert.pem /run/secrets/certificate
COPY key.pem /run/secrets/key

EXPOSE 5000

#The command use the same path for certificate and key
CMD ["flask", "--app", "app_test", "run", "--host=0.0.0.0", "--port=5000",
"--cert=/run/secrets/certificate", "--key=/run/secrets/key"]
```

In this example, the **COPY** puts the certificate and the key in the same path as using docker compose secrets **(/run/secrets/).**

Doing like this allow us to use the same code for testing mode without changing paths of secrets.

Data in Use: If the system is compromised, an attacker can read or change data stored in memory or used by the CPU. Before processing any data, we must protect the system from injection attacks.

Advanced defense: (In this course, we will focus on basic protection before data is processed.)

- Using hardware protections such as Trusted Execution Environments (TEEs) or enclaves.
- Encrypting sensitive data while it is stored in memory or being processed.
- Using advanced cryptographic methods like homomorphic encryption or Secure Multi-Party Computation (SMPC).

Basic defense: Input sanitization

Its goal is to protect the system from harmful input and injection attacks. In general, it means removing or filtering special characters from input strings.

You can do this manually or use libraries designed for sanitization.

Example: Run **microase** with docker compose and check the logs after sending a **GET** to:

`http://localhost:5000/str/reduce?op=upper&lst=print("ciao",flush=True)`

Sanitize input: There are tons of way to sanitize input. The basic strategy is to avoid unnecessary characters in the input and avoid to build output strings with unchecked input. **Easiest way:** type checking and regular expression

```python
import re

def sanitize_username(input_str):
    # Only allows alphanumeric characters, underscores, and hyphens
    sanitized_str = re.sub(r'[^a-zA-Z0-9_-]', '', input_str)
    return sanitized_str

# Example usage
user_input = "user@name!#$"
safe_input = sanitize_username(user_input)
print("Sanitized input:", safe_input) #Output: username
```

Check all the routes of our microservices and be careful when we accept input from query strings or payloads. Sanitize inputs by checking the type and excluding unnecessary characters. Avoid using or outputting unchecked data to other services or databases.

Data at Rest: If the system is compromised, databases data can be read and tampered with.

Basic defense: access control and encryption (again)

- Follow the Least Privilege Principle to give access to only who need it.
- You can encrypt at different level (db, table, field, etc.) The bigger the encrypted part the slower are performance. E.g. Encrypting the database means to en/decrypt data at every write/read

We have to choose carefully the data at rest to encrypt: It depends on situation. Security vs Performance is always a trade-off without a win-win.

Database technologies (Mongo, MySQL etc) provide support to access control and encryption.

- Access with credentials or API keys.
- HTTPS for communication
- Native encryption of data

DB: Access Control and Encryption

This part really depends on the DB technology used. You have to check the documentation to how enable credentials and encryption (it is also needed the certificate and the secret key).

Example:

MongoDB: You can see how secrets are used to pass credentials and TSL configuration.
The command is changed to enable authorization and HTTPS.

```
services:
…
  db:
    image: mongo:latest
    ports:
      - '27017:27017'
    volumes:
      - dbdata:/data/db
      - configdata:/data/configdb
    environment:
      MONGO_INITDB_ROOT_USERNAME: root
      MONGO_INITDB_DATABASE: admin
      MONGO_INITDB_ROOT_PASSWORD_FILE: /run/secrets/mongodb_password
    command: mongod --auth --quiet --logpath /dev/null
      --tlsAllowConnectionsWithoutCertificates
      --tlsMode preferTLS
      --tlsCertificateKeyFile /run/secrets/mongodb_certkey
      --tlsCAFile /run/secrets/mongodb_certkey
    secrets:
      - mongodb_password
      - mongodb_certkey

volumes:
  dbdata:
  configdata:

secrets:
  mongodb_certkey:
    file: mongo.pem
  mongodb_password:
    file: mongo_password.txt
```

In the example, you can see:

- **Authorization and credentials** (--auth, env variables and secrets)

- TLS (MongoDB wants a single file with certificate and private key)

Also, here (like with Postman) we disable certificate verification for teaching purposes, as MongoDB does not accept self-signed certificates.

DB Manager: Here the certificate and the password are taken from secrets and the URI contains TLS configuration. Obviously this configuration must match the db one.

```python
from pymongo import MongoClient

username = "root"
host = "db"
port = "27017"
db = "admin"
database = "my_database"

with open('/run/secrets/db_password', 'r') as file:
    password = file.read().strip()


uri =
f"mongodb://{username}:{password}@{host}:{port}/{database}?authSource={db}&tls=true&
tlsAllowInvalidCertificates=true"

client = MongoClient(uri)
```

DB: Encrypt Data

How to encrypt data depends on what and where you encrypt:

- **Fields:** encrypt/decrypt data in the DB manager before sending them to the DB.
- **Tables and database:** exploit the DB features where available (check their docs).

MongoDB does not support collection encryption natively. You have to encrypt/decrypt everything before the DB.

Example: Encrypt/Decrypt Data

Python example using Fernet as library for symmetric encryption. Ideally, we should used a KMS to managed keys

```python
from flask import Flask
from cryptography.fernet import Fernet

app = Flask(__name__)

# Generate a key for encryption and decryption
# This example creates a key every run
# We should read the key from secrets as before
key = Fernet.generate_key()
cipher_suite = Fernet(key)

@app.route('/save_data', methods=['POST'])
def save_data():
    data = request.json.get('data')
    # Encrypt the data
    encrypted_data = cipher_suite.encrypt(data.encode())
    # Store encrypted data
    db[data_id] = encrypted_data
    # Answer to the client on success
@app.route('/get_data/<data_id>', methods=['GET'])
def get_data(data_id):
    # Fetch encrypted data from the "database"
    encrypted_data = db.get(data_id)
    # Decrypt the data
    decrypted_data = cipher_suite.decrypt(encrypted_data).decode()

    return jsonify({"data": decrypted_data}), 200
```

Key Management Systems (KMS)

Systems to manage the cryptographic keys. Encryption is useless if you do not care about your keys. Not required for the project.

- Keys Exchange.
- Keys Storage.
- Keys Usage.
- Keys Rotation.

Lab:

✅ Lab 9: Security Analyses

Theory:

Secure Coding Practices

- More complex code usually contains more security problems.
- As the number of lines of code grows, the number of defects grows even faster.
- Both functional testing and security testing are essential to catch issues early.

There are two main types of testing and analysis:

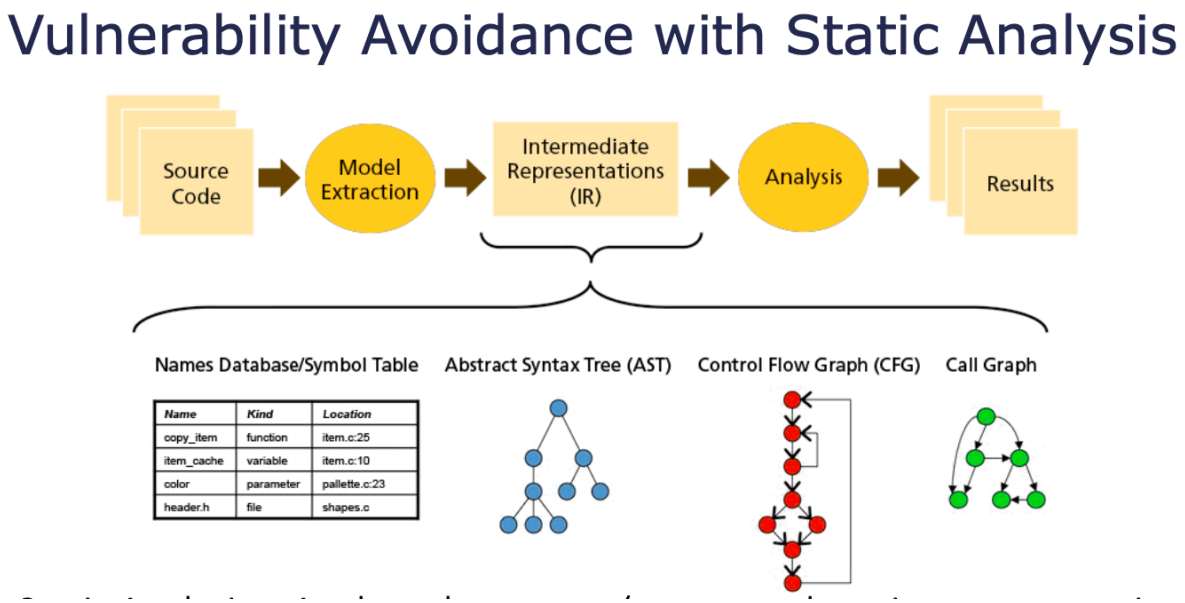
- **Static** (without running the program)
- **Dynamic** (by running the program)

Dynamic Analysis means running the application in a controlled environment and watching how it behaves in real time. It helps identify problems that only appear when the program is running, such as:

- Memory leaks
- Unintended data exposure
- Authentication or authorization issues

It is especially useful for finding **zero-day vulnerabilities** and unexpected behaviors caused by external conditions.

Static Analysis = Analyze the system (source code or its representation) to check some property without running it.



Bandit is a static analysis tool that checks Python code for common security problems. It works by matching your code against known insecure patterns using a set of plugins. It was first created for the **OpenStack Security Project**, and later moved to the **PyCQA** organization.

Bandit can detect **about 70 security issues** without any extra configuration. Documentation is available at: https://bandit.readthedocs.io/en/latest/

`pip install bandit`

`bandit -r <path to code>`

Example: [B324: hashlib]

```
>> Issue: [B324:hashlib] Use of weak MD5 hash for security. Consider usedforsecurity=False
   Severity: High   Confidence: High
   CWE: CWE-327 (https://cwe.mitre.org/data/definitions/327.html)
   More Info: https://bandit.readthedocs.io/en/1.7.5/plugins/b324_hashlib.html
   Location: ASE/lab6/test.py:2:9
1       import hashlib
2       result = hashlib.md5(b'ASE ASE ASE')
3       print("The byte equivalent of hash is : ", end ="")
```

- **Issue:** Use of MD5 (cryptographic hash function).
- **Severity:** of the Issue (how much the Issue is dangerous):
  - Bandit classifies issues in Low, Medium and High.
  - It gives a level of confidence (how much Bandit is confident about it) for every issue, also Low, Medium and High.
- **CWE:** Definition of why it is an issue. You can use it for understand how to resolve it.
- **More Info:** Bandit's info about the issue.
- **Location:** Location of the issue. In this example:
  - The path for the file having the issue is **ASE/lab6/**
  - The file with the issue is **test.py**
  - The line of the issue is 2 (and column 9).

How to resolve it? Change the hash function with a secure one.

From: `result = hashlib.md5(b'ASE ASE ASE')`

To: `result = hashlib.sha256(b'ASE ASE ASE')`

Remember:

- Do not change the behavior of the code! (in the example the hash of the string is done but with SHA256).
- Do not follow Bandit's suggestions blindly! (in the example the suggestion was to put `usedforsecurity=False`, ask yourself when it is correct).

Dependencies: pip-audit

pip-audit is a tool that checks your Python environment for packages with known security vulnerabilities. It uses the **Python Packaging Advisory Database** (through the PyPI JSON API) to look up reported issues. It scans all packages installed with **pip**.

Be cautious: fixing vulnerabilities may require upgrading packages, which can break backward compatibility.

pip-audit also has an official **GitHub Action** for CI pipelines.

Install: `pip install pip-audit`

Run: `pip-audit`

Fix vulnerabilities: `pip-audit —fix`

How it works in practice:

- pip-audit scans the Python environment on the machine (or container) where it runs.
- Since Python packages are often installed inside Docker images, you must run pip-audit inside those images if you want accurate results.

Ways to integrate it:

- Add the pip-audit command directly into the Docker image build.
- Create a dedicated Docker image that runs pip-audit before building the final images.
- Choose the approach that best fits your workflow.

Dependencies: GitHub Dependabot

Dependabot is a GitHub tool that checks your project's dependencies for known security vulnerabilities. How to enable it:

1. Open your repository **Settings**.
2. Go to **Advanced Security**.
3. Turn on **Dependabot security updates** and **Grouped security updates**.
4. Then open the **Security** tab of your repository to see if Dependabot has found any issues.

In most cases, if your project uses newer libraries, you will likely have no critical warnings.

Docker Images: Docker Scout

Docker Scout is a tool that scans Docker images for vulnerabilities. You need a Docker account for the easiest usage.

- It works with **local images** (on your machine) and **remote images** (in registries).

Documentation: https://docs.docker.com/scout/

Docker Scout in Docker Desktop: Docker Scout → Analyze Image:

View packages and CVEs → Fixable:

Resolve any (fixable) **critical** and **high** vulnerability. Some vulnerabilities are not fixable in this moment, you can filter by fixable vulnerabilities in the details. Most of the time you need only to update base images.

Docker Images: Trivy

Trivy is an all-in-one, open-source security scanner for cloud-native apps. Docs @ https://trivy.dev/docs/latest/ . It can scan: Container images, Local filesystems, Git repositories, Kubernetes clusters.

Our focus is on container (Docker) images. Trivy supports two targets for container images. Looking for vulnerabilities, misconfigurations, secrets, licenses.

- Files inside container images.
- Container image metadata.

Scan the image with: `trivy image <image-name>`

By default, it scans for vulnerabilities and secrets. You can specify what scan with the option: `--scanners`

Adding **vulns, misconfig**, or **license**. Secrets are always there.

Secrets are not Docker secrets. Are rules to find if are exposed:

- AWS access key, GCP service account
- (Github, GitLab, Slack) personal access token etc.

Lab:

Step 1: Install everything for the Lab.

Install Bandit: `pip install bandit`

Install pip-audit: `pip install pip-audit`

Install Trivy: `brew install aquasecurity/trivy/trivy`

Install Docker Scout: I will use Docker Desktop so no need to install anything.

Step 2: Run Bandit.

bandit -r src runned the bandit and gave the vulnerabilities of code.

```
Code scanned:
        Total lines of code: 369
        Total lines skipped (#nosec): 0

Run metrics:
        Total issues (by severity):
                Undefined: 0
                Low: 2
                Medium: 6
                High: 2
        Total issues (by confidence):
                Undefined: 0
                Low: 5
                Medium: 3
                High: 2
Files skipped (0):
```

We have fixed all the vulnerabilities of code, it was easy since Bandit shows everything, and where is exactly vulnerability located. Some example of problem and how I solved them:

```
Test results:
>> Issue: [B311:blacklist] Standard pseudo-random generators are not suitable for security/cryptographic purposes.
   Severity: Low   Confidence: High
   CWE: CWE-330 (https://cwe.mitre.org/data/definitions/330.html)
   More Info: https://bandit.readthedocs.io/en/1.9.2/blacklists/blacklist_calls.html#b311-random
   Location: src/calc/app.py:71:14
70                  return make_response('Invalid input\n', 400)
71              res = random.randint(a, b)
72              save_last("random", (a, b), res)
```

1. `B311:blacklist` — **insecure random generator**

Changed line of code `res = random.randint(a, b).` `random` is not cryptographically secure. If this “random” value has any security impact, Bandit warns you. to: `res = a + secrets.randbelow(b - a + 1)`

```
>> Issue: [B113:request_without_timeout] Call to requests without timeout
   Severity: Medium   Confidence: Low
   CWE: CWE-400 (https://cwe.mitre.org/data/definitions/400.html)
   More Info: https://bandit.readthedocs.io/en/1.9.2/plugins/b113_request_without_timeout.html
   Location: src/calc/app.py:130:8
129             payload = {'timestamp': timestamp, 'op': op, 'args': args, 'res': res}
130             requests.post('https://dbm:5000/notify', json=payload, verify='/run/secrets/dbm_cert')
131
```

2. `B113:request_without_timeout` — **missing timeout.** multiple places was missing timeout. Just added `timeout=3` looking in bandit reports.

Then I fixed everything related with these problems, and run again bandit and got response:

```
Test results:
        No issues identified.

Code scanned:
        Total lines of code: 369
        Total lines skipped (#nosec): 0

Run metrics:
        Total issues (by severity):
                Undefined: 0
                Low: 0
                Medium: 0
                High: 0
        Total issues (by confidence):
                Undefined: 0
                Low: 0
                Medium: 0
                High: 0
Files skipped (0):
```

Summary of fixes:

Issue	Fix
B311 insecure random	replace `random.randint` with `secrets.randbelow`
B113 timeout missing	add `timeout=3` to all requests
B201 debug=True	remove debug mode in Flask
B105 hardcoded password	use env vars or secrets
B307 eval()	replace with `ast.literal_eval()`

Step 2: Pip-Audit check

`docker compose run --rm (calc/dbm/string/gateway) sh` - created container image for the each service separately to try out them separately.
`pip install —upgrade pip` - inside the container i have updated pip version since it showed that new version is available
`pip install pip-audit` - installed again pip-audit dependencies inside the container since it wasn't available inside container.
`pip-audit` - then runed pip-audit command to check the services for vulnerabilities and bugs.

```
-lib, pip-audit
Successfully installed CacheControl-0.14.4 boolean.py-5.0 cyclonedx-pyth
.0.0 mdurl-0.1.2 msgpack-1.1.2 packageurl-python-0.17.6 packaging-25.0 p
lizable-2.1.0 pygments-2.19.2 pyparsing-3.2.5 rich-14.2.0 sortedcontaine
WARNING: Running pip as the 'root' user can result in broken permissions
m unusable. It is recommended to use a virtual environment instead: http
 doing and want to suppress this warning.
# pip-audit
No known vulnerabilities found
#
```

Got a reply with "No known vulnerabilities found"

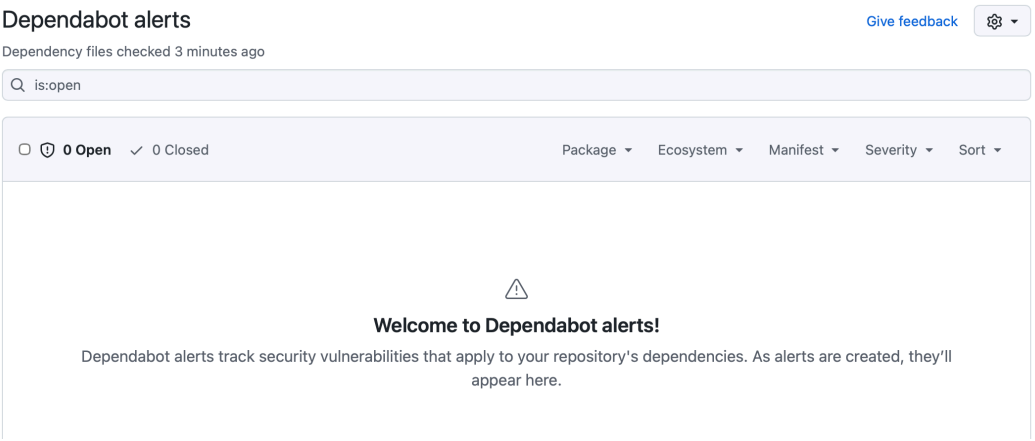
Step 3: Dependabot

Enable Dependabot. Inside GitHub repo: **Settings → Security & Analysis**

Enable:

- Dependabot security updates
- Dependabot grouped security updates

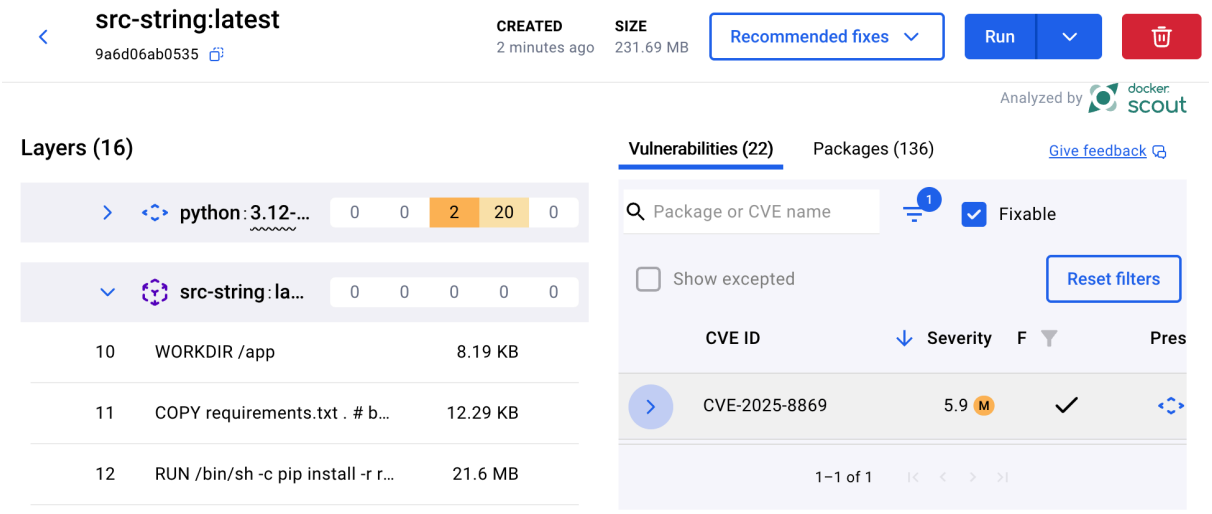
Then we checked it again with GitHub, which is type of pip-audit check, but again to make sure everything is okay. This time with Github Dependabot. Reply was:



Step 4: Docker Scout.

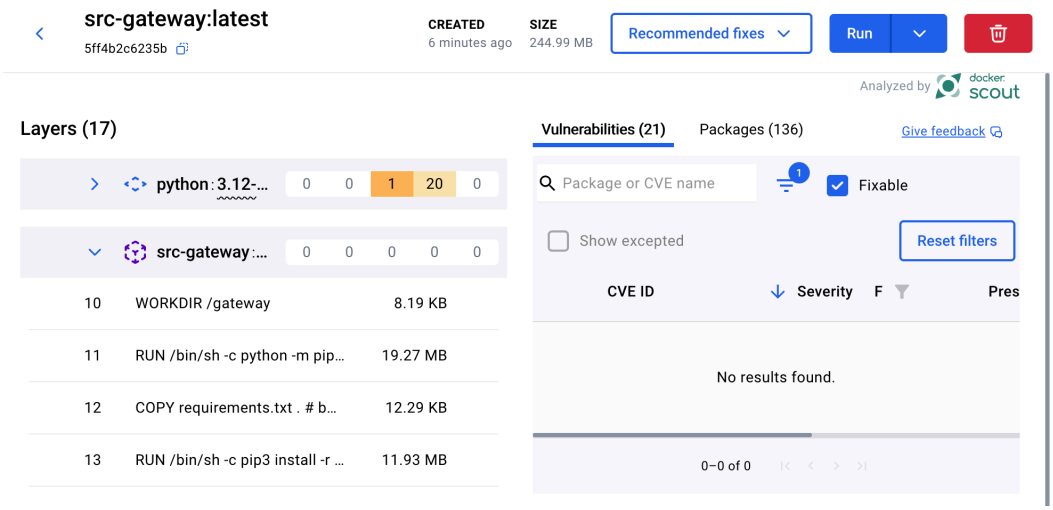
Before I have had to Run Docker Desktop. And of course build all the images with `docker compose build`. Then I have checked them with `docker images` in console.

Using Docker Scout was the easiest one. Since Docker Desktop is installed in my computer, I opened the **tab Docker Scout** → choose the **Docker Image** and **Analyze** it.



I had one kind of problem in every image, that was the pip-install version is not the newes, and that was the only medium problem. So I have decided to solve it in Dockerfile layers, adding:

`RUN python -m pip install --upgrade pip` to upgrade the version of pip before creating and running images and containers. Then this problem disapeared.



After that I didn't have problems, which was fixable by myself. There was only low affecting problems, which was only problem of base image and thats all.

Step 5: Trivy

Trivy was just completing check of Docker Scout. It was giving me the same result as Docker Scout. There wasn't also problems, rather than base image low affecting.

`trivy image <image-name>` this is the command to run the check with trivy
Lab was completed with this.

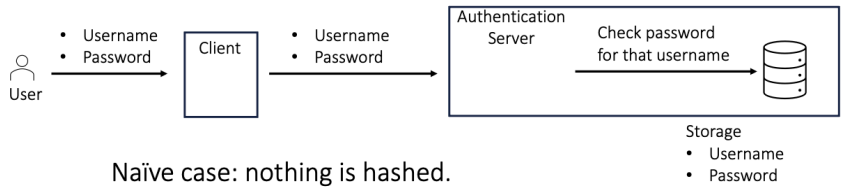
Lab 10: Access Security

Theory:

Access to the architecture

- Different APIs for 'general' users and 'special' users.
- Better if different networks, e.g. one public and one with a private VPN.

Credentials management:



Naïve case: nothing is hashed.
(Client and Server use HTTPS)

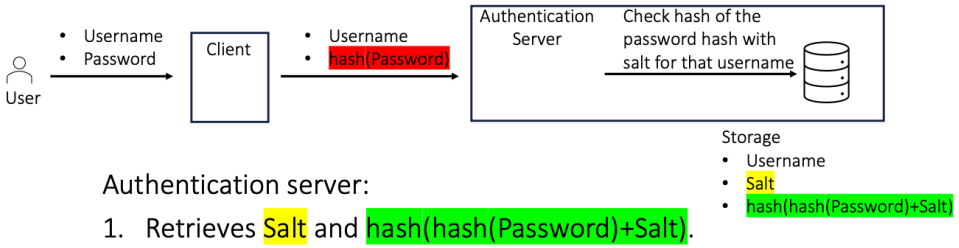
The Client: should hash the password. This avoid that the server knows a password for a user and try to use it for different service.

Q: Ho to send (Username, Hash Password) in REST?

A: As encrypted (HTTPS) payload.

Using query strings `(?user=aa&pw=aa)` is not safe! They can be cached by browsers, exposed via redirection etc.

The Authentication Server should store the password hashed with salt (a random value). This to make brute force and rainbow tables attacks harder.



Authentication server:

1. Retrieves Salt and hash(hash(Password)+Salt).
2. Applies hash to the received password with Salt.
3. Compare the hashed value with the retrieved one.

In general, services add another authentication channel (Multi-Factor Authentication), such as SMS, email, mobile app notification, etc. Password should be changed periodically.

For our purposes we only have to consider the server side. We do not care if the password arrives hashed, we store (hashed with salt) whathever comes.

Cryptographic hash functions (CHFs) have the same properties of hash functions plus:

- **Pre-image Resistance:** It's computationally infeasible to derive the original input from the hash.
- **Second Pre-image Resistance:** it's hard to find a different input that results in the same hash as a given input.

- **Collision Resistance:** It's hard to find two different inputs that produce the same hash.
- **Avalanche Effect:** A small change in input results in a significantly different hash. You can use libraries that hash data for you with secure algorithms.

Authorization – Oauth2.0

OAuth 2.0 is a standard framework that allows a third-party application to get limited access to an HTTP service on behalf of a user (the Resource Owner).

A **grant** is the credential that represents the user's authorization, such as a login session, username/password, or an ID token. OAuth 2.0 defines how authorization works (protocol and grant flows), but it does **not** define:

- how the user is authenticated,
- what token formats must look like,
- how the resource server should validate access tokens.

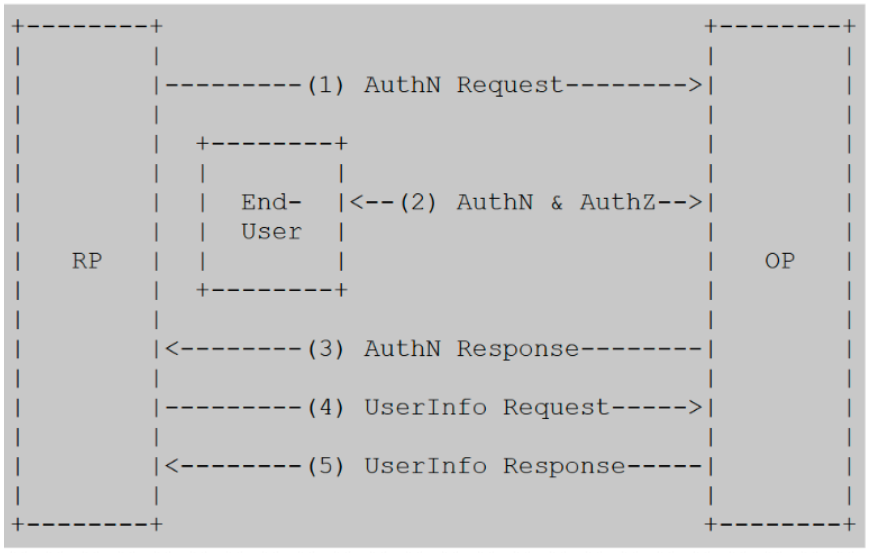
Authentication – OpenID Connect (OIDC)

OpenID Connect is an identity layer built on top of OAuth 2.0. It provides a standard way to authenticate users and to deliver identity information.

It lets users log in to multiple applications with one set of credentials. Its main addition to OAuth 2.0 is the **ID Token**, which carries verified information about the authenticated user.

OpenID Connect protocol:

1. The client (Relying Party) sends an authentication request to the OpenID Provider (OP).
2. The OP authenticates the user (usually via its web login page).
3. The OP sends back an **ID Token** and usually an **Access Token** to the client.
4. The client can use the Access Token to request additional user information from the **UserInfo endpoint**.
5. The UserInfo endpoint returns verified user data (Claims).



Tokens: There are two main types of tokens:

- **ID Tokens** – used for authentication and to access user information. They stay on the client side.
- **Access Tokens** – used for authorization. After login, they are sent with every request in the header: `Authorization: Bearer <access_token>`

Tokens should have short lifetimes to limit the risk of impersonation. Both OAuth 2.0 and OIDC include ways to refresh tokens when they expire.

Token format:

- OIDC uses **JSON Web Tokens (JWT)** for representing tokens.
- It is now common to use JWTs for authorization as well.

JSON Web Token (JWT) is a compact way to transmit information securely as a JSON object. It consists of three Base64-URL encoded parts separated by dots:

- Header
- Payload
- Signature (a hash of the encoded header and payload plus a private key)

Format: `Header.Payload.Signature`

The header usually contains:

- **alg** – the algorithm used for the signature (e.g., HS256, RSA)
- **typ** – the token type (usually “JWT”)

The payload contains **claims**, which may be:

- **Registered claims** (predefined): issuer (iss), expiration (exp), subject (sub), audience (aud), scope
- **Public claims** (defined openly)
- **Private claims** (custom application-specific data)

Important: A JWT is protected against tampering, but it is **not encrypted**, so anyone can read its contents. Sensitive or secret information must never be placed in the header or payload unless the token is encrypted.

Authorization in the system follows a simple sequence:

1. A user registers, and a record for that user is stored in the database.
2. When the user logs in, the system checks the provided credentials against the stored record.
3. If the credentials are correct, the server generates an **Access Token** and returns it to the user.
4. The Access Token contains a **scope**, which tells the system what microservices the user is allowed to access and who the user is (the subject).
5. For every API call, the user must include the Access Token so the system can verify whether the operation is permitted.

   Example: a user with the role **Math_user** can only access the APIs of the calculation microservice.

User login flow:

1. A registered user sends credentials to the gateway's login endpoint.

The gateway forwards the request to the authentication server's token endpoint.

2. The authentication server checks the credentials. If valid, it returns an Access Token.
3. For any later API call, the user must send the Access Token with the request.
4. If token validation is **centralized**, internal microservices must call the authentication server's **introspection endpoint** to verify the Access Token.

Token management (what the auth server must do):

- Create tokens during login.
- Validate tokens (in a centralized setup).
- Optionally refresh tokens.

To validate a JWT, the auth server (or internal services) must decode it using a JWT library and then check the signature, the subject, the scope, the expiration time, and other required claims.

Project: How to Test with tokens?

You must have at least one test user with test credentials. The easiest method is to create a dedicated Postman request that registers this test user. In theory, this request should be removed once testing is finished.

All your tests should:

1. Log in using the test credentials.
2. Get the authentication token.
3. Use that token to run the test operations.

In Postman:

- Create a Postman environment and add a variable (for example: **auth_token**) with an empty value.
- Use the test credentials to make a login request to get the token.
- In the *Tests* tab (post-response script) of the login request, write a script that reads the token from the response and saves it into the **auth_token** variable.

```
pm.test("Token received", function () {
    pm.response.to.have.status(200);
    var jsonData = pm.response.json();
    pm.environment.set("auth_token", jsonData.access_token);
});
```

- This will save the token in the environment variable. Every other request must use the environment variable

With locust (assuming the test user registered):

- Perform a login in the on_start() method that is called when each user is started.
- Save the token and add it to the header of each request.

What about testing of microservice in isolation?

1. If you use centralized authorization:
    - The token validation is an external dependency.
    - You have to mock it.
2. If you use distributed authorization:
    - Generate a mock token that is valid, or
    - Create mock code that skip the token validation. This will not test the authorization part in isolation but could simplify the things for you.

✅ Lab 11: Security Assessment

Theory + Lab:

Security in the software life cycle

- Security should be considered in every stage of building software.
- Improving security is an ongoing, repeated process because requirements and code change over time.
- The best approach is to include security from the very first version. If that is not possible, you can add it later by first reviewing the current state of the software.

Software Development Life Cycle (SDLC):

1. **Requirements:** define what the software must do.
    - Security requirements. Identify what security features the software needs, such as authentication, authorization, and data protection. Understand what assets (data, functions, systems) must be protected.
2. **Design:** plan how the software will work and how it will be structured.
    - Security architecture review. Look at the system's overall structure to see where it could be attacked. Decide which security principles to apply, such as defense in depth or least privilege.
    - Threat modeling and risk analysis. Identify possible threats to the system and determine the risks they create.
    - Attacker modeling. Describe what types of attackers might target the system, what they know, what resources they have, and how they might try to attack it.
3. **Implementation:** write the actual code.
    - Secure practices and code reviews. Follow established security best practices when writing code. Have other developers review your code to catch mistakes and security issues.
    - Static analysis. Use tools to automatically scan the source code for security problems (for example, tools like Bandit).
    - Dependency analysis. Check all external libraries and packages for known vulnerabilities (using tools such as pip-audit, Docker Scout, or Dependabot).
4. **Testing:** check that the software works correctly and securely.
    - Dynamic analysis. Check for vulnerabilities while the software is running.
    - Penetration testing. Simulate real attacks to find security weaknesses.
    - Fuzz testing. Send random or unexpected inputs to the system to see if it behaves incorrectly or fails.
    - Configuration review. Examine deployment scripts, Dockerfiles, docker-compose files, and other configuration files to ensure they are secure.
5. **Deployment and execution:** release the software and run it in the real environment.

- Monitoring. Continuously observe how the software behaves while it is running, for example by using intrusion detection tools.
- Incident response. Investigate urgent security issues and apply fixes as quickly as possible.

6. **Review:** evaluate the system and start the cycle again from step 1.
    - Collect information. Organize and collect data about not urgent problems, feedback from users and outputs of the monitoring
    - Next cycle planification. The collected information should be prepared for the next iteration, prepare them in terms of requirements, threats, etc.

Threat Modeling is a structured approach to identifying, understanding, and prioritizing potential security threats to a system. It helps developers think like attackers to anticipate risks and design protections before vulnerabilities are exploited.

It is is part of security assessment and answers question: What could possibly go wrong?

Threat modeling is similar to a risk assessment because it looks at what needs protection, what could go wrong, and how serious each risk is. But unlike a simple risk assessment, threat modeling also suggests ways to protect against those risks. It works by thinking like an attacker, so it focuses on the attacks that are most likely to happen.

When to perform threat modeling? During the design phase and whenever significant changes are made. It is an iterative process.

Threat modeling - Key Phases:

1. Work from a Model: Create a simple diagram or visual of your system so you can see its parts, how data moves, and how components interact. **Ways to do it:**

- **Data Flow Diagrams (DFDs):** Show how data moves through the system, including storage, processes, and outside connections.
- **Architectural Diagrams:** Show the main system structure and components.

Why: Helps spot areas where security issues could appear, like unprotected data paths or exposed services.

---

2. Identify Assets: Figure out what needs protection in the system. Assets are anything valuable, like data, infrastructure, or users. **Steps:**

1. List important assets such as sensitive data (personal info, payment details), APIs, or key components.
2. Rank them by importance to the business and the impact if they were compromised.

Example: User passwords and payment information are critical assets.

---

3. Identify Attack Surfaces: Find places where an attacker could interact with your system, like UI elements, APIs, or network endpoints. **Types:**

- **External:** Public areas like websites or public APIs.
- **Internal:** Areas only accessible to employees or trusted systems.

  Steps:

1. Look at all entry points (login forms, file uploads).
2. Check backend services (databases, microservices).

Example: A mobile app's attack surfaces might include the login screen, API endpoints, and app store deployment.

---

4. Identify Trust Boundaries: Mark points where data moves between trusted and untrusted areas. **Why:** These areas need extra protection, like authentication, validation, or encryption. **Steps:**

1. Use your system diagram to find trust boundaries.
2. Note protections at each boundary (firewalls, secure APIs).

Example: In an e-commerce platform, trust boundaries include user-to-application and application-to-payment gateway communication.

---

5. Identify Threats: List possible attacks against your system in a structured way. **How:**

- Brainstorm attack scenarios for your assets and attack surfaces.
- Use threat frameworks like STRIDE (Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, Elevation of privilege).

- Connect threats to specific components (e.g., SQL injection on a database).

Example: For an online shopping app: Spoofing: Fake logins to impersonate users.

---

6. Mitigate Threats: Plan ways to reduce the chance or impact of attacks.
Methods:

- **Preventative:** Stop attacks before they happen (input validation, access control).
- **Detective:** Detect attacks when they happen (logging, monitoring).
- **Corrective:** Reduce damage after an attack (backups, incident response).
- Prioritize based on risk.

Example: To prevent SQL injection, use parameterized queries and input validation.

Outcomes of Threat Modeling:

- **Proactive Defense:** Fixes and protections are put in place before attacks happen.
- **Better Understanding:** The team learns the system better and spots weaknesses.
- **Prioritization:** Focus on the most serious threats first to use resources wisely.

STRIDE (Threat Classification Model): is a method to classify security threats. Each letter stands for a type of threat:

- **S – Spoofing:** Pretending to be someone else.
- **T – Tampering:** Changing data or code without permission.
- **R – Repudiation:** Denying an action or transaction.
- **I – Information Disclosure:** Leaking sensitive data.
- **D – Denial of Service:** Making a service unavailable.
- **E – Elevation of Privilege:** Gaining higher access than allowed.