Ghent University

Cloud Applications and Mobile (E736010)

Lab 2: Web Development in Python: FastAPI

Professor:
Prof. dr.ir. Sofie Van Hoecke

Assistants:
Dr. Jennifer B. Sartor
Ing. Stef Pletinck

2024-2025

UNIVERSITEIT GENT

Table of contents

1 Introduction

In the previous lab we configured the RPi and set up a connection between your PC and the RPi using SSH. We accessed and read from sensors and actuators that are controlled remotely through the RPi. We ran Python code on the RPi with the easy-to-use `RPi.GPIO` library. Finally, we read and controlled sensors and actuators with an ESP32 as well.

The next step is to pick a framework on which the applications that will be created later on will run. FastAPI will be used as a web development framework for Python because it is designed to get someone started quickly and easily and has the ability to scale up to complex applications. You can see FastAPI listed in the Ingress box in Figure 1, which shows the architecture of the labs of this course. The techniques used in this lab session will get you ready for future lab sessions where the architecture starts to get more complex.

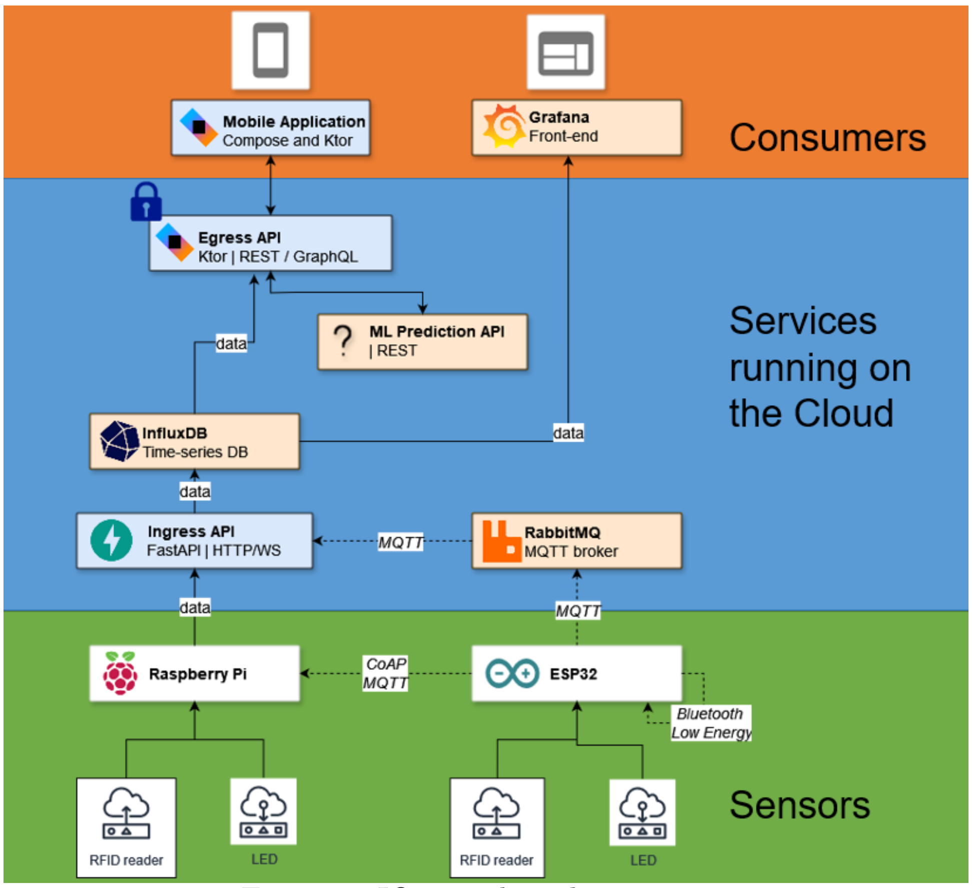


Figure 1: IOT-stack with categories

2 Goals

1. Set up virtual environments in PyCharm
2. Learn the basics of implementing the FastAPI library
3. Implement a registration system using FastAPI
4. Deploy a FastAPI application remotely on the RPi to control an actuator

3 Preparation

3.1 Tutorial: Python libraries

When installing Python packages, it is a good practise to do this inside a virtual environment.

3.1.1 Recommended: Virtual environment

It is recommended to use a virtual environment to manage the dependencies for your project. What problem does a virtual environment solve? The more Python projects you have, the more likely it is that you need to work with different versions of Python libraries or even Python itself. Newer versions of libraries for one project can break compatibility in another project.

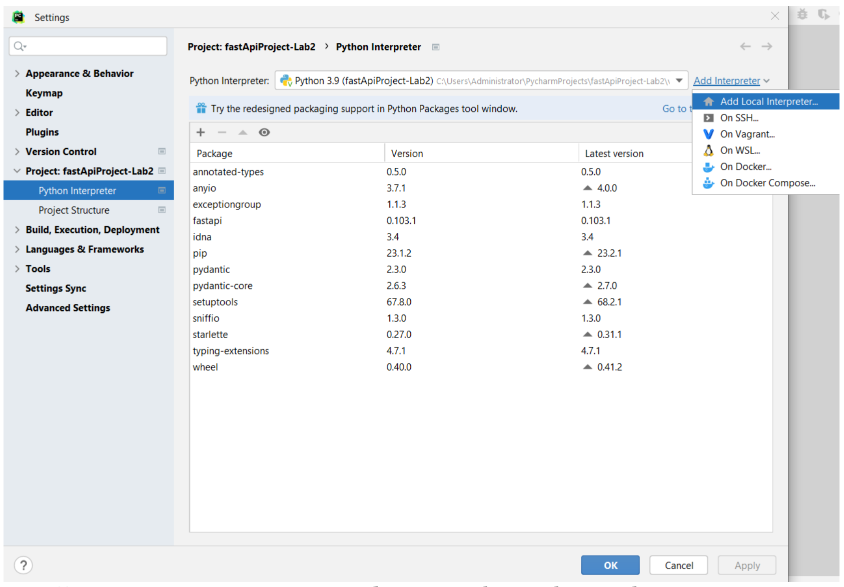
Virtual environments are independent groups of Python libraries, and each environment is coupled to a project. Packages installed for one project will not affect other projects or the operating system's packages. It is also possible to run FastAPI in a virtual environment as it is a Python package. The steps required to set up such a virtual environment are described below for Linux and Windows, as is how to use them in PyCharm.

The PyCharm IDE makes it easy for you as it creates and enters virtual environments automatically. The libraries and packages needed for each lab are always provided in a file called **requirements.txt**. PyCharm will help you install these within the virtual environment.

3.1.1.1 Virtual environment inside an existing project

First, open the code that we gave you on Ufora in PyCharm.

If you are inside an existing project, creating or changing the virtual environment can be done by going to the project interpreter. File > Settings... > Project: [Your Project name] > Project Interpreter



Project: fastApiProject-Lab2 › Python Interpreter

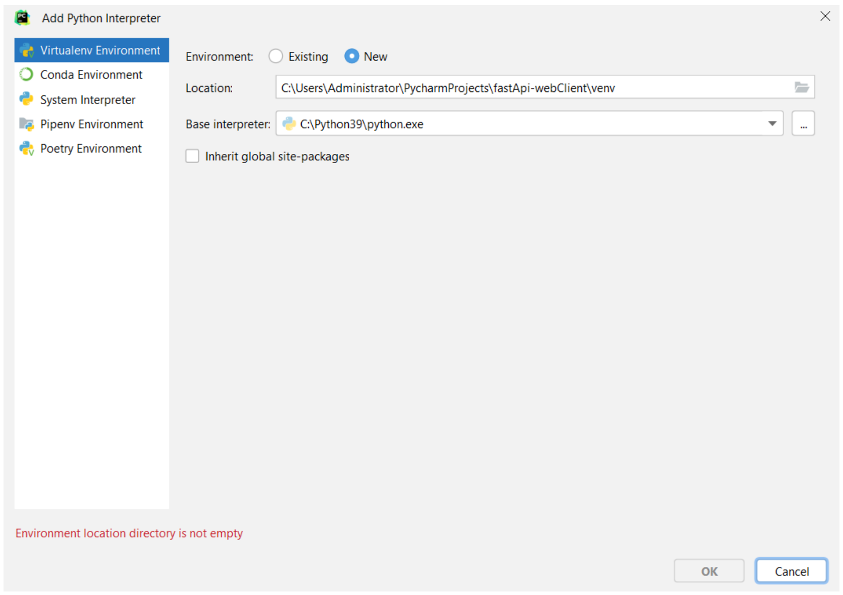
Python Interpreter: Python 3.9 (fastApiProject-Lab2) C:\Users\Administrator\PycharmProjects\fastApiProject-Lab2\

Try the redesigned packaging support in Python Packages tool window.

Package	Version	Latest version
annotated-types	0.5.0	0.5.0
anyio	3.7.1	▲ 4.0.0
exceptiongroup	1.1.3	1.1.3
fastapi	0.103.1	0.103.1
idna	3.4	3.4
pip	23.1.2	▲ 23.2.1
pydantic	2.3.0	2.3.0
pydantic-core	2.6.3	▲ 2.7.0
setuptools	67.8.0	▲ 68.2.1
sniffio	1.3.0	1.3.0
starlette	0.27.0	▲ 0.31.1
typing-extensions	4.7.1	4.7.1
wheel	0.40.0	▲ 0.41.2

Figure 2: Interpreter menu in order to select the right project interpreter

Create a virtual environment by clicking the "Add new interpreter" text next to your current project interpreter and add a new virtual environment with a name and Python 3.9, 3.10 or 3.11 (whichever you have installed), such as in Figure 3.



Environment: Existing [✓ New]

Location: C:\Users\Administrator\PycharmProjects\fastApi-webClient\venv

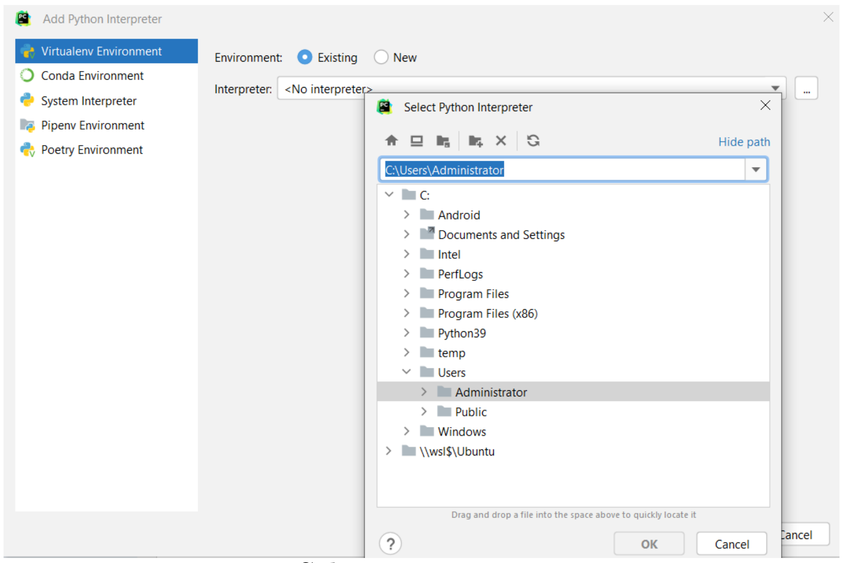
Base interpreter: C:\Python39\python.exe

☐ Inherit global site-packages

Environment location directory is not empty

Figure 3: Create new virtual environment and select Python version.

In PyCharm, you can also select an existing virtual environment with the file browser as seen in Figure 4.



Environment: [✓ Existing] New

Interpreter: <No interpreter>

C:\Users\Administrator

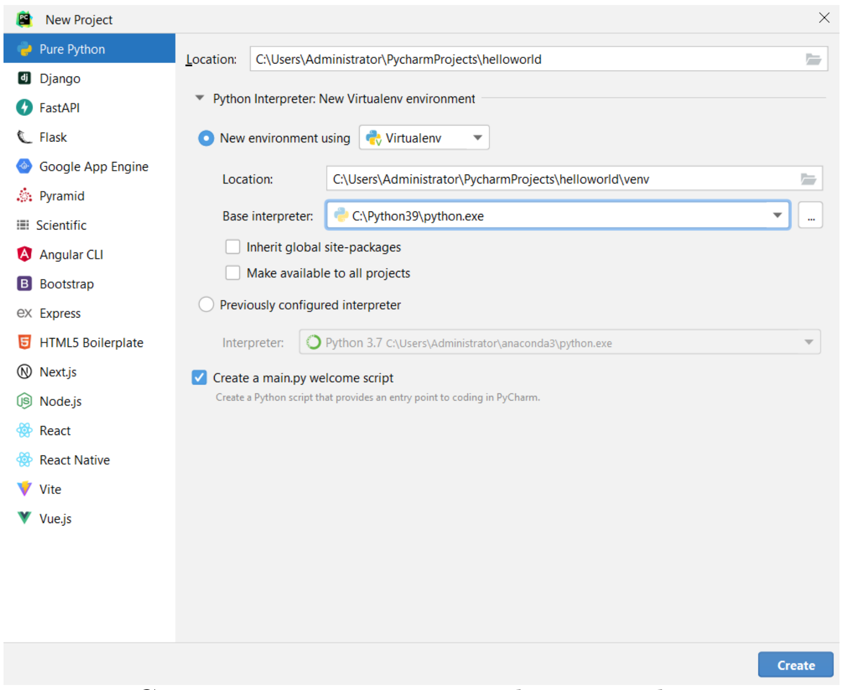
Drag and drop a file into the space above to quickly locate it

Figure 4: Select existing environment.

3.1.1.2 Making a New Virtual Environment

The steps to make a new virtual environment follow.

Make a new project under the Pure Python tab. Unfold the project interpreter and select a new environment using Virtualenv as seen in Figure 5.



Location: C:\Users\Administrator\PycharmProjects\helloworld

Python Interpreter: New Virtualenv environment

[✓ New environment using] Virtualenv

Location: C:\Users\Administrator\PycharmProjects\helloworld\venv

Base interpreter: C:\Python39\python.exe

Inherit global site-packages

Make available to all projects

Previously configured interpreter

Interpreter: Python 3.7 C:\Users\Administrator\anaconda3\python.exe

[✓ Create a main.py welcome script]

Create a Python script that provides an entry point to coding in PyCharm.

Figure 5: Creating a new project inside a virtual environment

3.1.2 If Virtual Environments Don't Work

If you want to install the Python dependencies directly on top of the Python version you currently have (only if virtual environments don't work for some reason), the next lines will help you out.

Use the following command to install FastAPI:
```
pip3 install "fastapi[all]"
```

The requirements needed for this lab are in **requirements.txt**. Installing these requirements can be done with the following command:
```
pip3 install -r requirements.txt
```

3.2 Upgrade RPi

Before we go on to a more complicated FastAPI application, make sure you have **upgraded your RPi's software at home** (which can take around half an hour) using `sudo apt-get update` and `sudo apt-get upgrade`. It's important to do this at home so you don't have to lose time during the lab itself. It is also important that the **time on your RPi is correctly set** before you update and upgrade. See the appendix of the write-up of Lab1 for how to do this.

4 First FastAPI application

In this section we will go over the code of a rather simple FastAPI application that we provide to you, and we will demonstrate some of the built-in functionality and extensions provided by

FastAPI so you are familiar with them. The idea is that we are going to build a web application that allows a user receiving their RFID tags for the first time to register their name together with their RFID ID. Then an organization who gives these RFID tags out would have a record of who has what RFID tag. Once they have registered, they can control an LED connected to our RPi. We give you the code to get started on this application, and the code that we provide you can run on your laptop with a new virtual environment that you created as explained in Section 3.1.1.1. Note, however, that **the code (and the associated requirements.txt libraries) needs to run on the RPi when you implement the LED part of this lab**. Therefore, when you come to Task2, you will have to use an ssh interpreter to run the code on the RPi. Before running the code, go through the following sections that explain the code that we give you. Note that the `requirements.txt` file installs both the fastapi and uvicorn packages. The uvicorn package is a server that serves our API.

4.1 Application structure

It is very important when creating larger applications to have a good structure, so we give you an overview of a FastAPI project's structure in this section. This FastAPI app (`fastApiStarterStudent-` is provided on the online learning platform. Our FastAPI projects will generally have the following file structure:

- static/
- templates/
- main.py
- routes.py
- *models.py*

The Python file in *cursive* will be explained below. Here we go over some of the basic functionality in these files. The static folder holds files such as CSS and JavaScript files that your web applications might need for formatting. HTML templates reside in the templates folder, and we will use the Jinja2 template engine, which will be explained below.

```
from fastapi import FastAPI, APIRouter
from routes import router
from fastapi.staticfiles import StaticFiles

api_router = APIRouter()
api_router.include_router(router, prefix="", tags=["register"])

app = FastAPI()
app.include_router(api_router)
app.mount("/static", StaticFiles(directory="static"), name="static")
```

Listing 1: `main.py`

In Listing 1, we see the `main.py` file. We import the FastAPI class and an APIRouter from fastapi. We set up the APIRouter and include the router created in our `routes.py` file, which is discussed below. Then we create a FastAPI instance, include the router, and configure the directory where the static files can be found.

```
from typing import Optional
from fastapi import APIRouter, Request
from fastapi.templating import Jinja2Templates

templates = Jinja2Templates(directory="templates")
router = APIRouter()

@router.get("/")
def index(request: Request, alert:Optional[str] = None):
    return templates.TemplateResponse("index.html",
                                      {"request":request, "alert": alert})

@router.get("/hello/{name}")
async def say_hello(request: Request, name: str, alert:Optional[str] = None):
    return templates.TemplateResponse("index.html", {"request":request,
                                                     "name":name, "alert": alert})
```

Listing 2: `routes.py`

You can see `routes.py` in Listing 2. This is the file where we set up the various endpoints of our web application. First you can see that we set up our Jinja2 template engine to use the templates provided in the templates directory. We also set up a router in this file. Then the endpoints are defined.

First we create a path operation, or endpoint. The decorator `@router.get("/")` specifies that the function below will handle requests to the HTTP method `get` from a new endpoint after the domain name at "/". Below the decorator comes the function that handles those get requests to "/", and returns the message we want to display in the user's browser.

The second function defines another decorator for a different path, which uses a variable `name`. We can accept variables in a path and use these to personalize our application. In the code, the name entered after `/hello/` in the URL will be displayed on the page.

To understand what is displayed when someone sends a GET request, we will first explain templates.

4.2 Templates

FastAPI allows us to use the **Jinja2** template engine. Jinja2 allows us to quickly reuse a layout across multiple web pages instead of having to copy/paste the same code multiple times. An example `layout.html` is shown below in Listing 3:

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Lab 2 | {% block title %}{% endblock %}</title>

    <!-- Bootstrap 4.1.3 -->
    <link rel="stylesheet"
    ↪  href="../static/dist/bootstrap-4.0.0/css/bootstrap.min.css">
    <!-- Narrow Jumbotron -->
    <link rel="stylesheet"
    ↪  href="../static/dist/bootstrap-4.0.0/css/jumbotron-narrow.css">
    <!-- Page specific styles -->
    {% block head %} {% endblock %}
</head>
<body>
<div class="container">
    <div class="header clearfix">
        <nav>
            <ul class="nav nav-pills float-right">
                <li class="nav-item">
                    <a class="nav-link active" href="/">Home <span
                    ↪  class="sr-only">(current)</span></a>
                </li>

            </ul>
        </nav>
        <h3 class="text-muted">Labo 2 | {{ self.title() }}</h3>
    </div>

    {% block body %} {% endblock %}

    <footer class="footer">
        <p>UGent | Cloud and Mobile applications</p>
    </footer>

</div> <!-- /container -->

<!-- Javascript scripts -->
{% block scripts %} {% endblock %}
</body>
</html>
```

Listing 3: `templates/layout.html`

This template provides a basic layout for several other templates in our application, e.g. `index.html` in Listing 4. Note the different blocks that can be adjusted to your preferences.

```
{% extends 'layout.html' %}
{% block title %}My First FastAPI App{% endblock %}

{% block body %}
    <div class="jumbotron">
        {% if name %}
        <h1 class="display-3" style="overflow: hidden">Hello, {{ name }}!</h1>
            <p>Want to register? <a href="../register">Go to the registration
            ↪  page!</a></p>
            <p>Want to control a LED? <a href="../control_led">Go to the
            ↪  Control Room!</a></p>
        {% else  %}
            <h1 class="display-3">Hello, World!</h1>
            <p>Want to register? <a href="../register">Go to the registration
            ↪  page!</a></p>
        {% endif %}
    </div>

    <div class="row marketing">
        <div class="col-lg-12 text-center">

        </div>
        {% if alert %}
            <div class="alert alert-info" role="alert">
                {{alert}}
            </div>
        {% endif %}
    </div>
{% endblock %}
```

Listing 4: `templates/index.html`

In the `index.html` file, you can see an if/else statement which adds additional logic to handle an optional passed `name` variable. Jinja2 allows us to easily create interactive content.

Looking back at `routes.py` in Listing 2, we can now understand that, after a get request to the root ("/") endpoint, the template engine will render the given index.html, giving it no `name` variable, and passing it a `Request` and potentially an alert. The `Request` is needed simply because the html template renderer needs access to the `Request`, which is filled with the information about the request, such as headers, IP address, URL, etc. If there is an alert (used for error handling), there is code to display it in the index.html file.

By default, these endpoints will return HTTP status 200. This can be customised by passing the `status_code` argument in the decorator.

Any arguments to your endpoint functions will automatically be checked and passed to the function. In the code above, we define an argument `name`, which we also define in the route string, e.g. "@router.get("/hello/{name}")". This is what FastAPI calls a path parameter. When you surf to `http://127.0.0.1:8000/hello/Jennifer` when your webapp is running, the string "Jennifer" will be passed into your `say_hello` function.

When you declare other function parameters that are not part of the path parameters, they are automatically interpreted as "query" parameters. The query is the set of key-value pairs that go after the '?' in a URL, separated by '&' characters. By default, any arguments are taken from the query string, which is the part after the question mark in a URL, e.g.

`"http://127.0.0.1:8000/hello/Jennifer?alert=Successfully Logged In"`. These are called query parameters.

JSON request bodies can also be injected into arguments, such as can be seen in `routes.py` above. For example:
`def index(request: Request, alert:Optional[str] = None):`
In the above method definition, the first parameter is a request. The second, optional `alert` parameter is a query parameter as explained above. All of these parameters will have their types checked, so any missing parameters or errors, for example, a string where an int is expected, will return a 422 error to the user. If you want to understand more, see FastAPI's Tutorial and User Guide.

An example of running this basic webapp, http://<Your RPi IP here>:8000/, in your browser can be seen in Figure 6. You can load this code in PyCharm, create a new virtual environment for it, and run it to test this yourself.

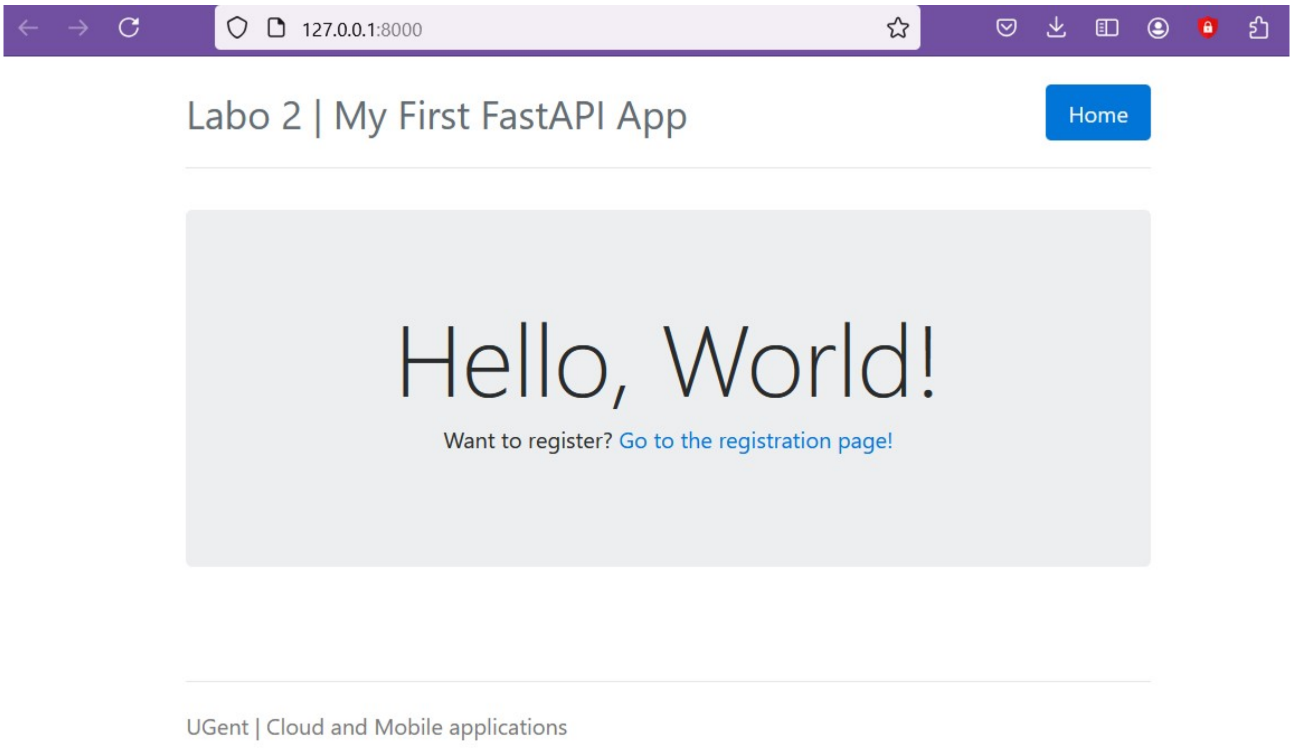

127.0.0.1:8000
Labo 2 | My First FastAPI App
Home
Hello, World!
Want to register? Go to the registration page!
UGent | Cloud and Mobile applications

Figure 6: Index page of our web application.

4.3 Extensions beyond the basic app

The parts of the application that we have explained so far consist of the most basic functionalities FastAPI offers; these will be implemented in just about every FastAPI application that you might write. Aside from this, a lot of extensions can be added on top of this basic application thanks to FastAPI being an open-source project with an active community.

In addition to the files listed above, we have another file that is specific to this project because we want to control an LED connected to the RPi. The additional file is called `MyRPi.py`. This file is very similar to the code you wrote in Lab1, and implements a python class to control an LED connected to the RPi, including get the status of the LED, and to turn the LED on or off. You will implement this class yourself in one of the tasks below.

To understand the `models.py` file, we first have to know that there is code (partially) provided in `routes.py` that implements two endpoints to display a registration page and to post a new registration attempt by a user. Only after registration can a user go to the page to control the LED.

First, we will look at two classes defined in `models.py`:

```python
from typing import Annotated
from dataclasses import dataclass
from fastapi import Form

class RFIDuser:
    def __init__(self, firstname, lastname, email, rfidid):
        self.firstname = firstname
        self.lastname = lastname
        self.rfidid = rfidid
        self.email = email

    def __str__(self) -> str:
        return "Name: " + self.firstname  + " " + self.lastname + ", email: "
        +  self.email + ", RFID ID: " + self.rfidid

@dataclass
class Registration:
    firstname: Annotated[str, Form()]
    lastname: Annotated[str, Form()]
    email: Annotated[str, Form()]
    rfidid: Annotated[str, Form()]
```

Listing 5: From `models.py`, the classes `Registration` and `RFIDuser`.

The class `Registration` defines the fields we expect when someone registers with our web application. When someone registers, they must provide their first name, last name, unique email address and RFID ID.

We only require a few routes or endpoints to be able to achieve our goal of registration. See Listing 6 to see those endpoints that we have defined in `routes.py`.

```python
from models import Registration, RFIDuser

users = []

@router.get("/register")
def show_registration(request: Request, alert:Optional[str] = None):
    return templates.TemplateResponse("register.html", {"request": request,
                                                        "alert": alert})

@router.post("/register")
def register(request: Request, data: Registration = Depends(),
             alert:Optional[str] = None):
    # TODO: check if user in users list, then
    # return index template
```

Listing 6: Code snippet from `routes.py` that performs registration.

We have two functions defined to support registration. One handles an HTTP GET request, and the other the POST. The show_registration function just renders the registration page via the template. Just like the endpoint for "/" defined above, the show_registration function accepts a Request and an optional alert parameter. The register function handles a POST request and takes an extra parameter. The data that is expected as a parameter to the function is defined by the imported Registration class from models.py. Pydantic can do error checking to make sure the fields are all filled in and are of the right type, etc. The data parameter is defined with Depends. FastAPI has a powerful dependency injection system that allows your path operation functions to declare things that it requires to work and use, using Depends. FastAPI takes care of providing the method with those needed dependencies.

You will write the body of the register function in a task below. We created an empty list, users, that will hold RFIDusers, or the class seen above in models.py. For simplicity, we leave out the implementation to store users' information in a database. When a user tries to register, you should first check if they already exist in the local list, provide an appropriate alert, and render the index page with the correct fields. You will also have to extend the given register.html template, which is shown in Listing 7. The html code has a form that will call the corresponding POST method when submitted, and currently only includes one field for the first name of the user. The registration page displayed in the browser should look like Figure 7.

Note that if you look in main.py, you will see a method handle_validation_alerts, which turns validation exceptions into alert parameters. This method applies to every route. If a validation exception occurs, this method will turn it into an alert which can be displayed by the html pages.

```html
{% extends 'layout.html' %}
{% block title %}Registration{% endblock %}

{% block body %}
    <div class="jumbotron">
        <h1 class="display-3">Registration</h1>
    </div>

    <div class="row marketing">
        <div class="col-lg-12">
            <form method="POST" action="register">
                <input style="margin-bottom: 5px;" type="text"
                ↪  class="form-control" name="firstname" id="firstname"
                      placeholder="firstname">
                <button class="btn btn-primary">Register</button>
            </form>
        </div>
    </div>
{% endblock %}
```

Listing 7: register.html

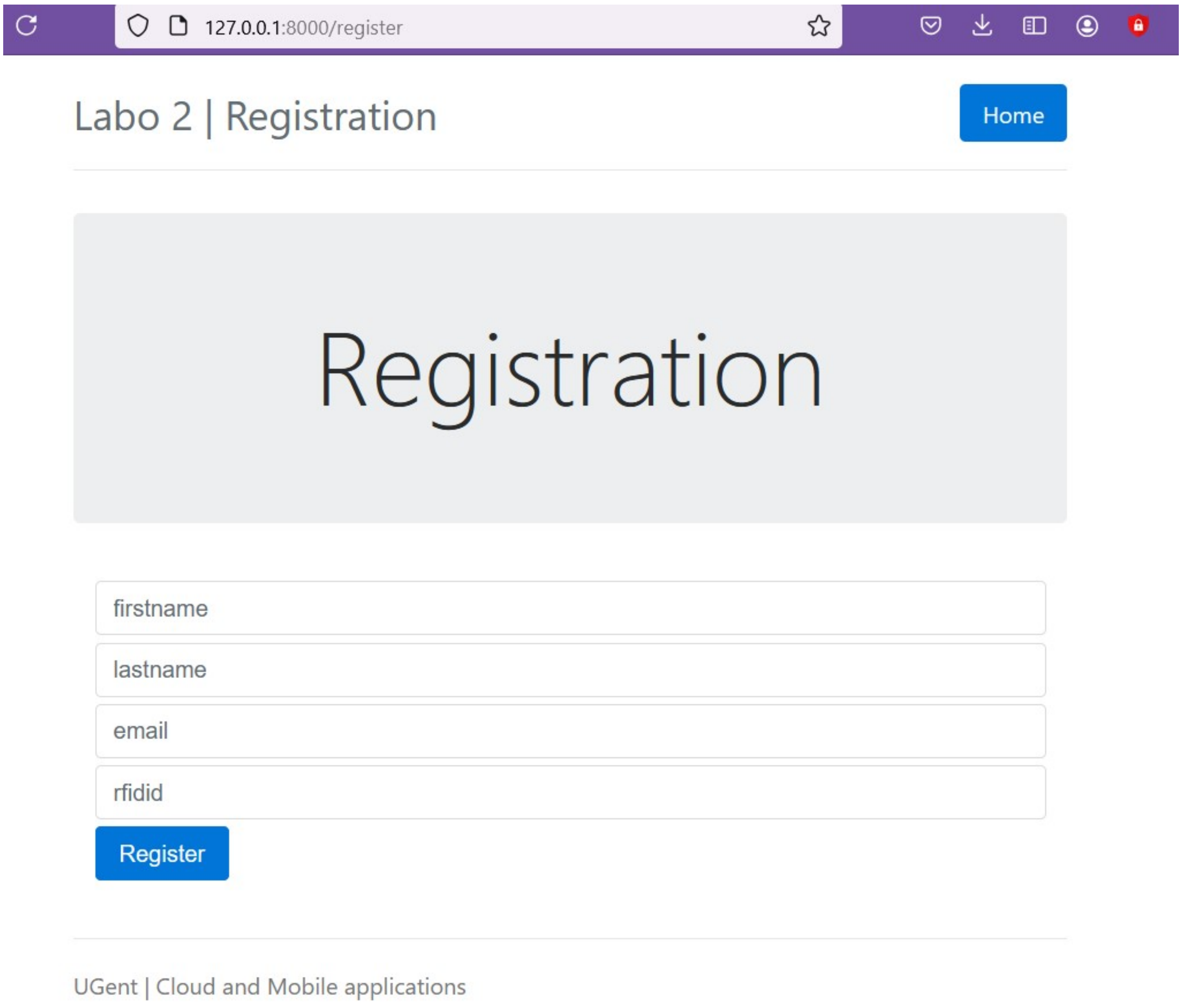


Figure 7: Registration page of our web application.

5 Tasks

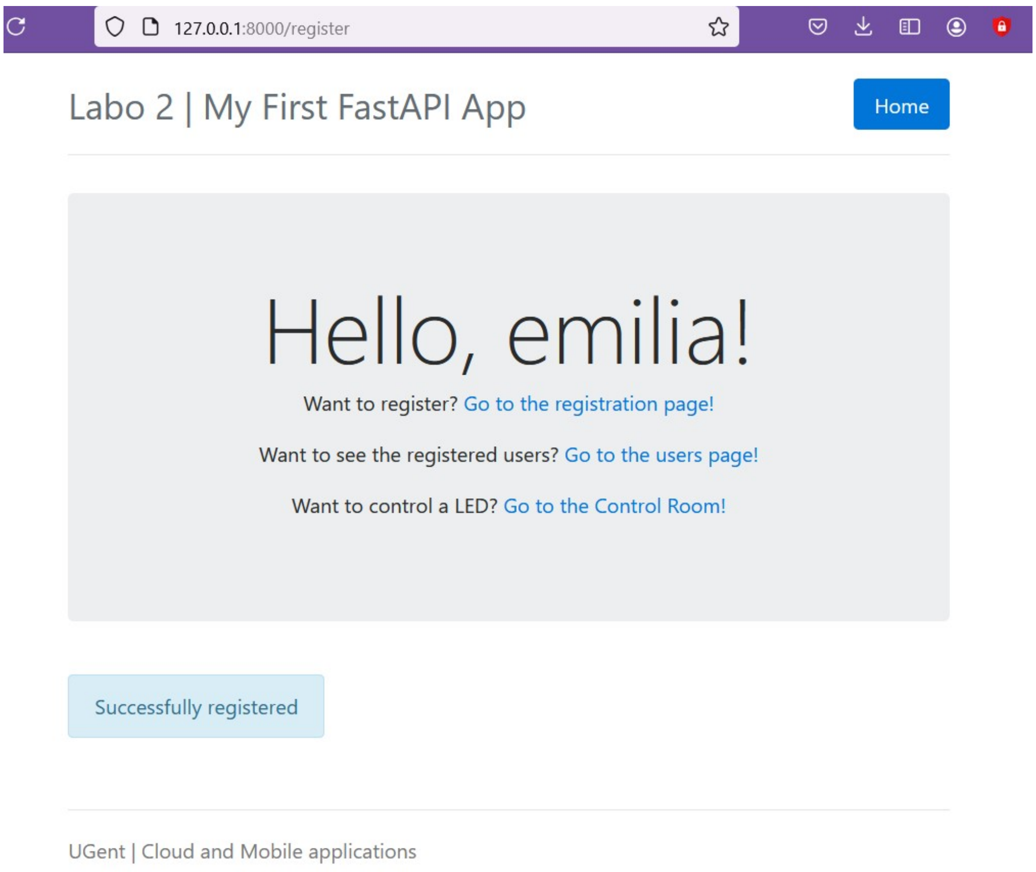
Don't forget to take **screenshots or short movies** of the results of each task to show that you got the task working. You'll need to turn these in with your lab report.

5.1 Task 1: Registration

For this task, you have to complete the registration code, as explained above.

You need to modify the following files: `register.html` and `routes.py`. In Figure 7 the various fields that need to be filled in are specified, namely first name, last name, email address, and RFID ID.

If you succeed with registration you should be forwarded to the index webpage, which should look like Figure 8, except the line about going to the users page, which is an optional task explained below.



127.0.0.1:8000/register

Labo 2 | My First FastAPI App

Home

Hello, emilia!

Want to register? Go to the registration page!

Want to see the registered users? Go to the users page!

Want to control a LED? Go to the Control Room!

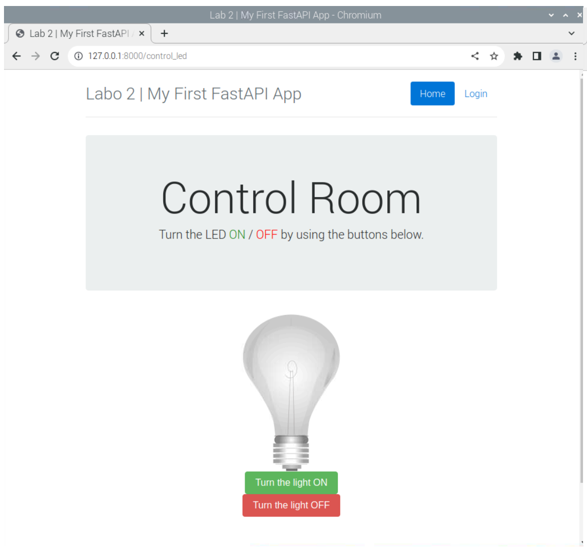
Successfully registered

UGent | Cloud and Mobile applications

Figure 8: Task 1 index page after user has registered.

5.2 Task 2: IoTRPi exercise

In this exercise you need to complete the code for your FastAPI application to turn an LED on and off via a webpage `http://<Your RPi IP here>:8000/control_led` using the `RPi.GPIO` library on the RPi as seen in the first lab session. After a user registers, they see a link to get to this page to control the LED. **This code has to run on your RPi and thus you have to switch to using an ssh interpreter.**



Lab 2 | My First FastAPI App - Chromium

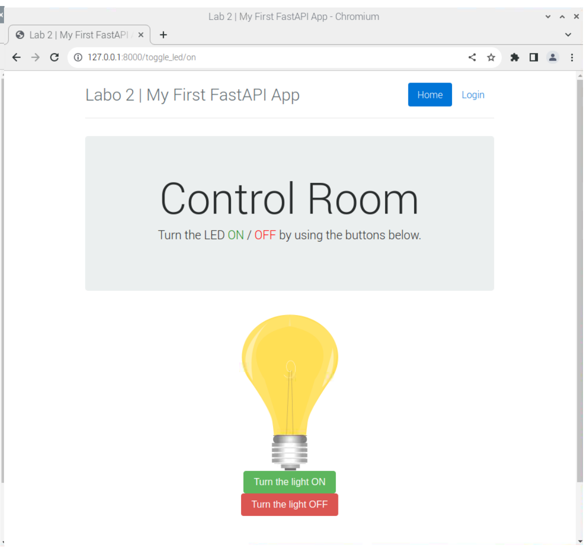
127.0.0.1:8000/control_led

Labo 2 | My First FastAPI App

Control Room

Turn the LED ON / OFF by using the buttons below.

Figure 9: Light off



Lab 2 | My First FastAPI App - Chromium

127.0.0.1:8000/toggle_led/on

Labo 2 | My First FastAPI App

Control Room

Turn the LED ON / OFF by using the buttons below.

Figure 10: Light on

Complete all the methods of the class `MyRPi`.

```python
import RPi.GPIO as GPIO

LED_PIN = 4


class RaspberryPi(type):
    """
    Implementation of Singleton pattern
    """
    _instances = {}

    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super(RaspberryPi, cls).__call__(*args,
             ↪  **kwargs)
        return cls._instances[cls]


class MyRPi(metaclass=RaspberryPi):
    """
    Class that is responsible for accessing and setting the pins on the RPi
    """
    def __init__(self):
        """
        Set the PIN mode to BCM (GPIOx)
        Initiate the LED_PIN as OUTPUT and initial value of LOW
        """

    def get_status(self):
        """
        Method to retrieve the status of the actuator
        :return: status as boolean (HIGH / LOW)
        """

    def set_status(self, status):
        """
        Method to set the status of the actuator
        :param status: boolean (HIGH / LOW)
        """
```

Listing 8: `MyRPi.py`

Complete the functions and their routing in the file `routes.py`. The `show_control_led` function to implement a get operation is listed for you, but you must also write a `control_led` function to handle a post request. Note that you have to pass the status of the LED at this moment to the .html file so you know which image to display of a light bulb.

```python
@router.get("/control_led")
def show_control_led(...):  # fill in parameters and body

# also function control_led needs to handle a post operation
```

Listing 9: Functions for you to implement in `routes.py`

Use the code snippet below to determine the code for the HTTP POST, and the code you need to fill in within `templates/control_led.html`. Note that we have provided images to use to display the lightbulb on that webpage in the static folder.

```
<form action="/toggle_led/on" method="post">
    <button name="LED" value="ON" class="btn btn-success" type="submit">Turn
     ↪  the light ON</button>
</form>
```

Listing 10: Code snippet to help you with `templates/control_led.html`

If you want to test if your web application works in a browser on your laptop and not only on your RPi (where the code is running), you can edit the configuration on PyCharm to expose the application to the local network rather than localhost (the machine itself) only. Otherwise the web application is only running on the localhost on the RPi and is not visible externally. First, go to the project name at the top right of PyCharm and click on the down arrow, as seen in Figure 11. Then click on 'Edit Configurations'. A new window will appear, as in Figure 12. In the 'Uvicorn options' text box, add the line '- -host 0.0.0.0'. Then click 'Apply' and 'OK', and you should be able to access something like 192.168.0.29:8000 via your laptop's web browser.

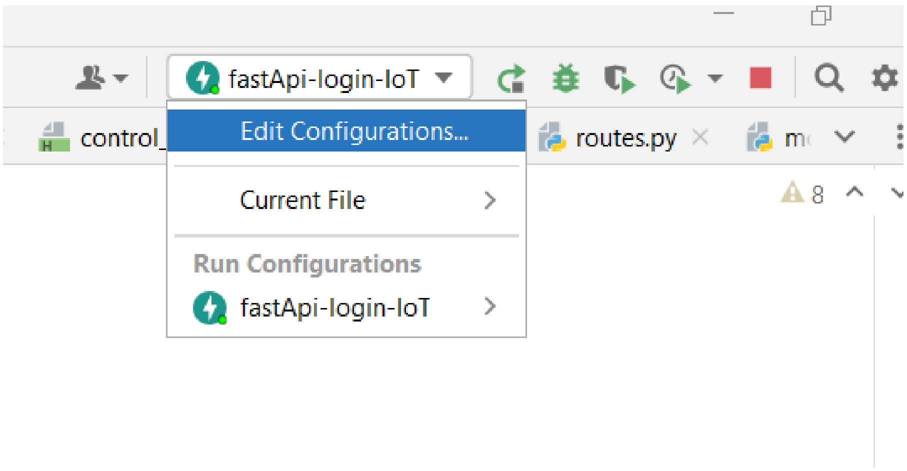
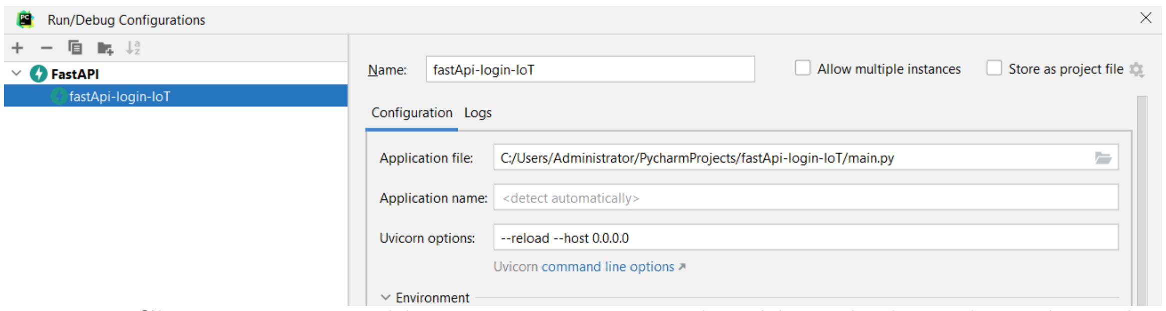


Figure 11: PyCharm: How to edit the configuration options.


Run/Debug Configurations

FastAPI
fastApi-login-IoT

Name: fastApi-login-IoT
Allow multiple instances
Store as project file

Configuration Logs

Application file: C:/Users/Administrator/PycharmProjects/fastApi-login-IoT/main.py
Application name: <detect automatically>
Uvicorn options: --reload --host 0.0.0.0
Uvicorn command line options ↗

Environment

Figure 12: PyCharm: How to add a uvicorn option to be able to look at the web application locally.

5.3 Optional Task 3: Visualize the RFID users on a new route

Do not start with this task unless you have done the previous tasks! Solving this last task correctly is meant for students that want to distinguish themselves.

Extend task 1 by visualizing the users list on a new route. Create an additional `HTML` page that uses the newly defined route.

You could also add a button to this new webpage to export these users to a file so that the users would persist after the webapp has been shut down.

6 Questions

1. How would you implement task 2 in such a way that a user would not have to refresh the page to see the current status of the LED if switched by another client?
2. Describe what happens and why when someone posts invalid data, such as a login with blank username, or a typo in the field name.

7 Summary

The powerful FastAPI library offers a lot of flexibility as we do web development in Python. Making use of various useful extensions on top of FastAPI, we have a framework on which we can quickly build a lot of functionality into our web applications.

8 Material to submit

Give the lab report a structure that mimics the structure of this document. If you had to do anything noteworthy in the preparation or "First FastAPI Application" sections, or had major problems, include this in your report. In your lab report, elaborate on how you solved the tasks along with an explanation of the code that you wrote. Include **screenshots** in your report or **videos** in your zip to show that you completed each task successfully. Do not forget to answer the questions in Section 6 as these are graded as well. Only the **.pdf** extension is allowed for the report. Archive this lab report together with your source code, name it **Lab2_FamilyName_FirstName.zip** and upload it to the online learning platform.