VRIJE
UNIVERSITEIT
BRUSSEL

ADVANCED WEB APPLICATION

Final project report

Valentin Quevy

2022-2023

Kris Steenhaut
Steffen Thielemans

ENGINEERING SCIENCES

Executive summary

This report is part of the final project for the master course "Advanced Web-Applications" at the Vrije Universiteit Brussel (Industrial Sciences, Electronics-ICT, Networks). The main goal of this course is to familiarize students with current technologies used for developing advanced web applications.

With this project, we were able to create an interactive web-app for remotely controlling and monitoring a set of lamps.

Contents

Chapter 1

Introduction

How do we develop modern web-applications ? In this report I will describe the way we developped one.

In chapter 2, we will go over everything that was discussed in class and list all of the technologies that we used in this project. This includes four major sections: networking, front-end, back-end, security, and threats. It will also be discussed why these technologies were chosen.

In chapter 3, I will describe how I implemented all of these technologies and the difficulties we encountered. A comprehensive overview of the project will be given. The backend of the web-app is the primary focus.

Finally, we will draw some conclusions and make some recommendations.

Figure 1.1: The classroom with the connected lamps

The sections that follow will discuss everything that was covered during class. I chose to start with the networking section, explaining all what happens at a higher level and providing a top-level view of the communication.

Then I'll go over what happens on the client and server sides. Finally, I discuss the security of the website and potential threats.

2.1 Networking

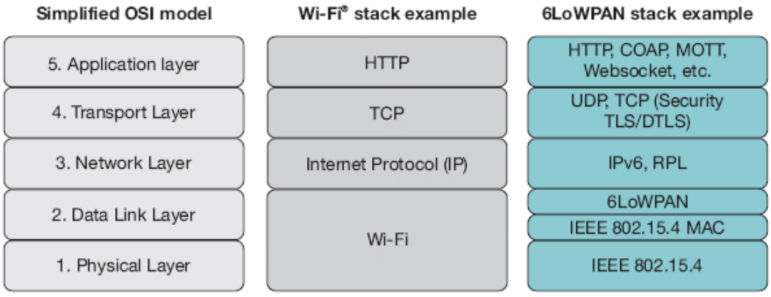

Figure 2.1: The OSI model, a Wi-Fi stack example and the 6LoWPAN stack exemple, source: https://www.researchgate.net/figure/The-OSI-model-a-Wi-Fi-stack-example-and-the-6LoWPAN-stack-exemple-13_fig3_332430222

2.1.1 CoAP client/server communication

The lamps we have to control are all equiped with Zolertia boards, these are considered IoT devices. These **constrained devices** are often implemented in lossy networks.

Constrained Application Protocol (CoAP) is an application layer protocol explicitly designed for those devices limited in ressources. This protocol is efficient because of it low overhead. CoAP easily translates to HTTP, making the web-compatibility possible. HTTP and CoAP share the REST model, they are characterized by their **statelessness** and separate the concerns of client and server. The clients are sending requests to retrieve or modify resources, and servers send responses to these requests. GET, POST, PUT and DELETE are the 4 types of requests the client can make. The server automically respond to these requests with response codes (200:OK, 201:created, 204:no content, 400:bad request, 403:forbidden, 404:not found, 500: internal server error).

2.1.2 Websockets

Previous protocols provide a request/response communication model and are half-duplex. WebSockets allow both the server and the client to push messages at any time (full-duplex) without any relation to a previous request. This is achieved by initiating an TCP-connection through an HTTP-handshake. This is an ideal protocol for near real-time communication.

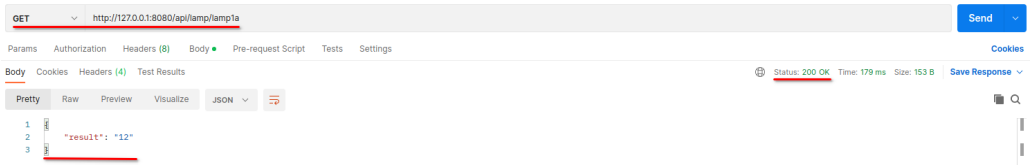

Figure 2.2: example of an HTTP GET-request response with postman

For our webapp it could be used to frequently refresh the lamp status at the client-side while the connection remains open.

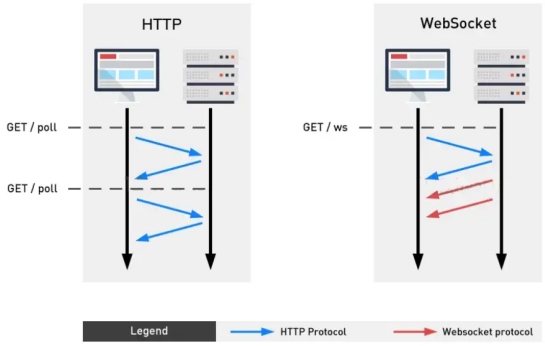

Figure 2.3: WebSockets allow the client-server connection to remain open, whereas traditionally the connection is closed after the server responds to a client request.

2.1.3 Docker containerization

Once the webapplication coded we have to deploy it. One way to deploy it would be to run it directly on a computer. This make it possible to be in conflict with the other packages on the computer or block the computer if the app fails.

Docker adds here an abstraction layer between the computer and the app by providing a lightweight and standalone OS-level virtualization.

On docker we first build our Docker image, this defines a snapshot of the environment. Finally we run an instance of the image, packaging all the code and all its dependencies into an container. With this it becomes possible to make very scalable modern webapplications by managing multiple containers independently.

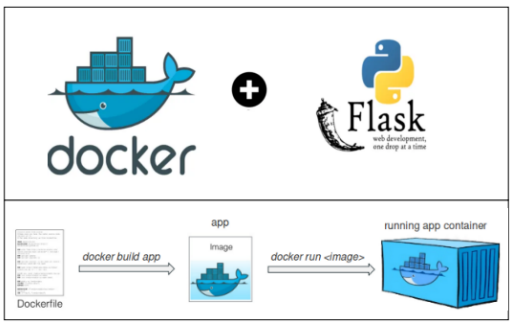

Figure 2.4: Containerization process with Docker, source: https://bikramat.medium.com/dockerfile-python-flask-e03a3c0dfe65

2.2 Client-side development

HTML, CSS, and JavaScript are the three languages used on mostly all websites for the User Interface. JavaScript is the programming language that allows the website to be responsive; HTML is used to structure the site; and CSS is used to design and layout the web page.

2.2.1 bootstrap

Building good websites also comes with providing a good UI and User Experience. As an engineer our first focus lays more on the working of the application and less on the layout of the UI, this often results in well working websites with poor design and not intuitive to use.

Bootstrap is built on these three languages and offer developers a pre-defined grid system. This makes it possible for building websites without spending to much time on the design.

2.3 Server-side development

2.3.1 Flask as an RESTful API

Flask is a web application framework written in Python that allows you to easily develop web applications. It is very light-weight and easy-to-extend core.

The requests send by our clients are handled by our Application Programming Interface respecting the REST standard. Depending on the type requests made and body an adequate response is returned.

Since Flask lacks several functionality out of the box, such as database management, asynchronous compatibility, CoAp, etc., additional modules must be imported.

2.3.2 Extra python modules

Flask-socketio gives our application acces to bidirectional communication through a permanent connection with our server hosting the app.

Asyncio makes it possible to start coroutines, control subprocess, distribute tasks via queues and synchronize concurrent code.

Aiocoap together with Asyncio will be used to communicate asynchronously with our nodes through the CoAP interface for monitoring and controlling the lamps.

Flask-login provides user session management for Flask: logging in, logging out, and remembering our users' sessions.

Bcrypt provides well-trusted hashing utilities for our application.

SQLAlchemy, for effective and high-performance database access offers a complete set of well-known enterprise-level persistence techniques. For logging our users, we'll utilize it along with a SQLlite database file.

2.4 Security and threats

Before saving sensitive information like passwords in a database it is primordial to not store them directly as plaintext strings but first salt and hash them. The process showed on fig. 2.5 makes it possible to securely save a password.

A **salt** is random string that is used as an additional input to a one-way function that **hashes** data, a password, or a passphrase cryptographically. Salts are used to protect passwords while they are being stored. Historically, only the output of a cryptographic hash function on the password was stored on a system, but as time passed, additional safeguards were developed to

prevent duplicate or common passwords from being identified (as their hashes are identical like in Rainbow tables).

SQL injection is a web security vulnerability that allows an attacker to interfere with database queries made by an application. In general, it enables an attacker to view data that they would not normally be able to access. SQLAlchemy is used to protect us against it by checking the input.

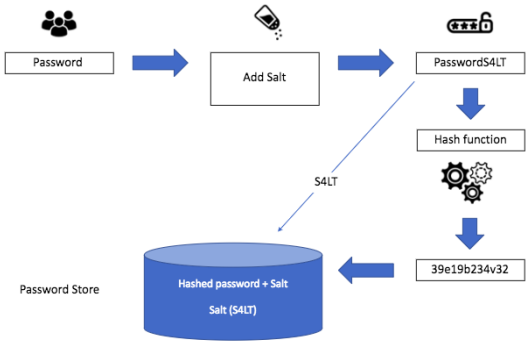

Figure 2.5: A secure way to store a password into a database through salting and hashing, source: https://www.okta.com/blog/2019/03/what-are-salted-passwords-and-password-hashing/

3.1 Setup

The goal of this project is to be able to monitor and control lamps via CoAP using a web-based application.

To access the lamp-nodes, we must first connect to our university WiFi before reaching a linux server (raspberry pi) that acts as a gateway to the nodes. Figure 3.3 depicts the project setup and communication links.

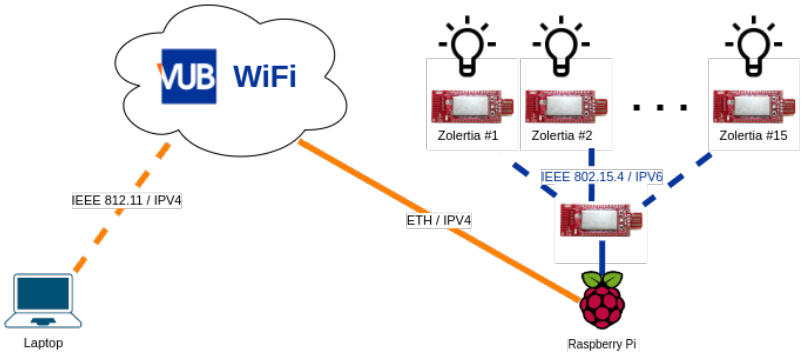

Figure 3.1: Project setup

3.2 Front-end

We cannot overlook the UI development of our online application. Websites with poor user interfaces frequently have little traffic and frustrated users.

Our web application's initial layout and design were generated through HTML and CSS. A quick approach for building a user-friendly UI was to adopt Bootstrap.

The login page for our online application is an adaptation of our university portal. The additional pages were made in order to meet the project criteria. A separate page was also made to display the lamp logs.

The pages are still basic because the server-side is where this project is mostly focused.

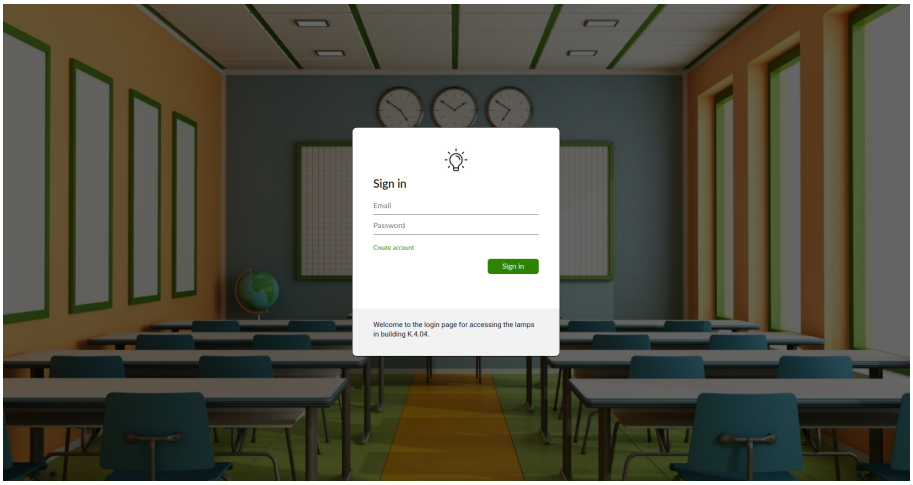

Figure 3.2: Login page

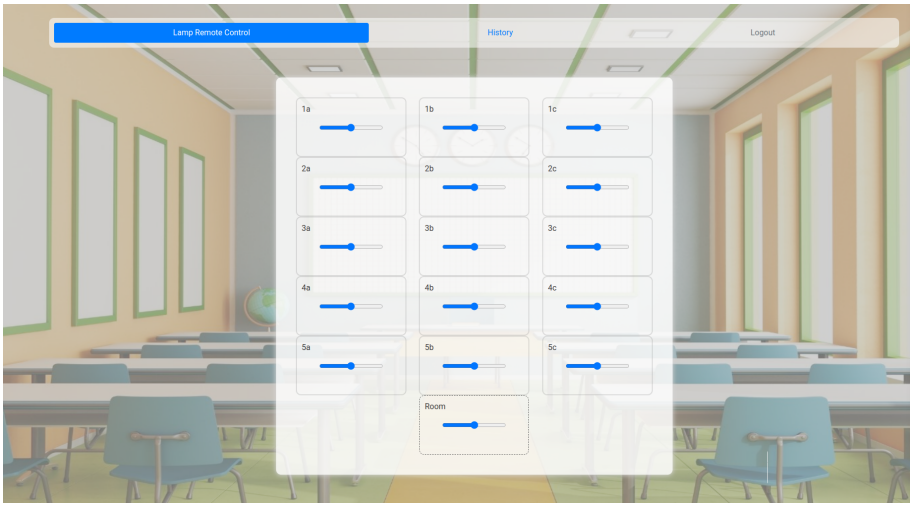

Figure 3.3: Lamp dashboard page

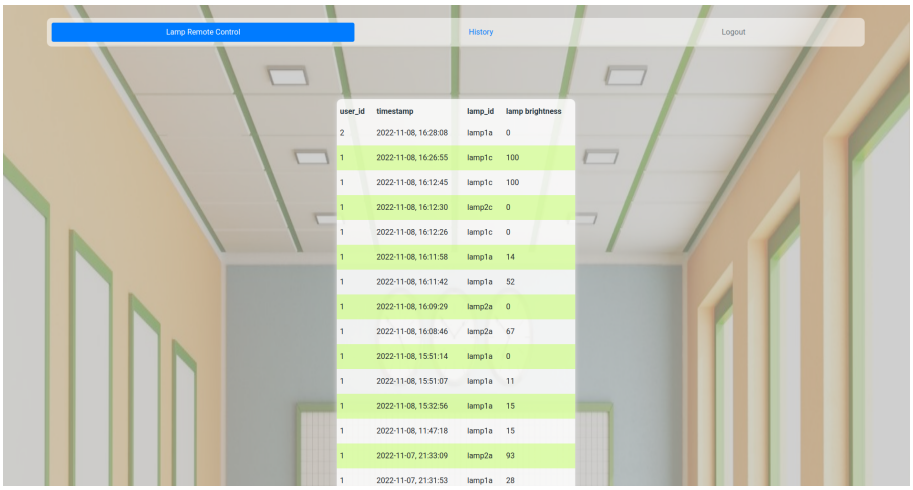

Figure 3.4: Lamp logs page

3.3 Back-end

Our server must respond to HTTP requests that are sent by our clients. The clients in this scenario are the people who use our web application and access it through specific hyperlinks.

3.3.1 Code description

This is all made possible by the Python scripts that are running on our server.

Our web application will process and reply to client requests in both an asynchronous and synchronous manner using **ASGI**, or the Asynchronous Server Gateway Interface. ASGI is the main file, instantiating our web-application and forwarding those requests to it. It also handles the web-socket communication. I put in place a background thread that would broadcast the lumunosity value of each lamp to the connected clients at predetermined intervals via the sockets because the lamps' lumunosities were constantly changing during class.

Listing 3.1: ASGI code, the main script running on our server

```
from webapp import create_app            # App constructor
from config import config                # App configurations
from coapclient import get_lamps_level   # Lamps level getter through CoAP
from flask_socketio import SocketIO      # Socket implementation
import threading                         # Launching background-thread

# constructing app based on the configuration
app = create_app(config)
# app with socket support
app_with_socket = SocketIO(app, async_mode='eventlet', debug=config.DEBUG)

def background_lamp_status(delay)->None:
    """"""
    while True:
        # sleeping before sending our next request
        app_with_socket.sleep(delay)
        # asynchronous function getting the lamps brightness
        lamps_level = get_lamps_level()
        # sending the lamps brightness through the socket to our clients
        app_with_socket.emit('ReceiveFromChatServer', lamps_level)

if __name__ == "__main__":

    with threading.Lock():  # launching periodic background thread
        app_with_socket.start_background_task(background_lamp_status, 5)

    app_with_socket.run(app, host='localhost', port=8080, debug=True,
    use_reloader=False) # running our app and making it accessible
```

Previous script imports numereous other scripts fulfilling various purposes.

Coapclient is the script provided by our class instructor for communicating through CoAP to our lamp-nodes. The main requests are getting and setting the lamp brightness.

Config is the file containing the configuration parameters of our web-application. Inline configuration is a bad coding style and dangerous. It is important to store the configuration parameters in a separate file as they can be regularly changed. This also prevents committing sensitive values such as secret keys in the middle of our codebase.

Webapp initialisator will define how our app has to be created. The **Extensions** is used to prevent circular imports and contains our database object, login-manager and hashing utilities. The **Handlers** script handles the login-manager behaviour.

Routes is the script defining the url's our web-application is pointing to. I wrote all the routes in this file since the there are only 7 url's. See appendix A for api documentation.

Models defines the tables of our database. I created 2 tables: User and LampLog.

Forms contains the users input like the login- and registerform. They define the the user input format and faclitate the rendering of it.

3.3.2 App container

I choosed to containerize the application in order to isolate the app from his environment. This makes the deployment of the application on various machines easy.

For this we first write a Dockerfile in order to build the application image. In this file we describe the packages needed and the commando's our container has to run for launching the app. In this case we need Python, the used modules and the software we wrote inside the container. We also have to specify the commando launching the application and have to expose the port to make the website reachable on our computer.

Once the image build we can run our container and access the application.

Conclusions and recommendations

Flask is an excellent tool for developing small and lightweight web applications. It is very extensible, but some modules were not always easy to implement. Asynchronous modules were occasionally in conflict with one another because they are not using the same protocol.

For larger projects, **Django**, which comes with more features out of the box than Flask, may be worth considering. If you want to stay in the Flask environment I recommend using the **Blueprint** module which architectures the application in sub-applications, making it very modular. For small application like this one it is unnecessary since we only have 7 url's.

Having had some troubles with previous similar projects, I recommend using **git** for having version control of the codebase, allowing you to go several steps back when the app fails.

Github is also useful for sharing code with classmates in in order to help them.

I also recommend using an **Virtual Environment** in order to only manage the Python packages of this project rather than using the global system Python packages, which may cause conflicts with other modules.

API-documentation

Authentification

`GET /login`

- Description
  Return login page
- URL Params:
  None
- Body Params:
  None
- Responses:
  - Success respone (200):
    "GET /sign-in HTTP/1.1" 200

`POST /login`

- Description
  Logs user in.
- URL Params:
  None
- Body Params:
  - email:string
  - password:string
- Responses:
  - Redirection (302):
    - if password is correct, redirects to `/`
      "POST /login HTTP/1.1" 302
      "GET / HTTP/1.1" 200

`GET /register`

- Description
  Return register page
- URL Params:
  None
- Responses:
  - Success respone (200):
    "GET /register HTTP/1.1" 200

`POST /register`

- Description
  Creates user account.
- URL Params:
  None
- Body Params:
  - email:string
  - password:string
- Responses:
  - Redirection (302):
    - if inputs filled, it creates the account and redirects to `/login`
      "POST /register HTTP/1.1" 302
      "GET /login HTTP/1.1" 200

`GET /logout`

- Description
  Logs user out.
- URL Params:
  None
- Body Params:
  None
- Responses:

  - Redirection (302): logs user out and redirects to `/login`

    ```
    "GET /logout HTTP/1.1" 302
    "GET /sign-in HTTP/1.1" 200
    ```

Lamp API

`GET /api/lamp/<lamp_id>/`

- Description
  Return lamp luminosity in json format (require login)
- URL Params:
  - lamp_id:string (example:lamp1a)
- Body Params:
  None
- Responses:
  - Success respone (200): Return value of specified lamp
    "GET /api/lamp/ HTTP/1.1" 200

`POST /api/lamp/<lamp_id>/`

- Description
  Set the lumunosity of specified lamp (require login)

- URL Params:
  - lamp_id:string (example:lamp1a)
- Body Params:
  None
- Responses:
  - Success respone (200): Set brightness of specified lamp
    "POST /api/lamp/ HTTP/1.1" 200

`GET /api/lamp/all/`

- Description
  Return lamp luminosity of all the lamps in json format (require login)
- URL Params:
  None
- Body Params:
  None
- Responses:
  - Success respone (200): Return value of all lamps
    "GET /api/lamp/all HTTP/1.1" 200

`POST /api/lamp/all/`

- Description
  Set the lumunosity of all lamps. (require login)
- URL Params:
  None
- Body Params:
  - dimming:integer
- Responses:
  - Success respone (200): Set brightness of all the lamps
    "POST /api/lamp/all HTTP/1.1" 200

Other pages

`GET /`

- Description
  Return the lamp dashboard page. (require login)
- URL Params:
  None
- Body Params:
  None
- Responses:
  - Success respone (200):
    "GET / HTTP/1.1" 200

`GET /history`

- Description
  Return the lamp history page. (require login)
- URL Params:
  None
- Body Params:
  None
- Responses:
  - Success respone (200):
    "GET /history HTTP/1.1" 200

Bibliography

Build Flask Apps Series (n.d.). https://hackersandslackers.com/series/build-flask-apps/. (Accessed on 11/10/2022).

DockerFile Python Flask. Dockerlize a Python Flask Application | by Bikram | Medium (n.d.). https://bikramat.medium.com/dockerfile-python-flask-e03a3c0dfe65. (Accessed on 11/10/2022).

Introduction | Socket.IO (n.d.). https://socket.io/docs/v4/. (Accessed on 11/10/2022).

Lamkimel, M., Naja, N., Jamali, A., and Yahyaoui, A. (Nov. 2018). 'The Internet of Things: Overview of the essential elements and the new enabling technology 6LoWPAN'. In: pp. 142–147. DOI: 10.1109/ITMC.2018.8691271.

Overview | Docker Documentation (n.d.). https://docs.docker.com/get-started/. (Accessed on 11/10/2022).

What are Salted Passwords and Password Hashing? | Okta (n.d.). https://www.okta.com/blog/2019/03/what-are-salted-passwords-and-password-hashing/. (Accessed on 11/10/2022).