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.

Chapter 1

Introduction

add intro-line

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 go over the results and make some conclusions and recommendations.

Chapter 2

Background and context

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.

rewrite sections and add pictures

2.1 Networking

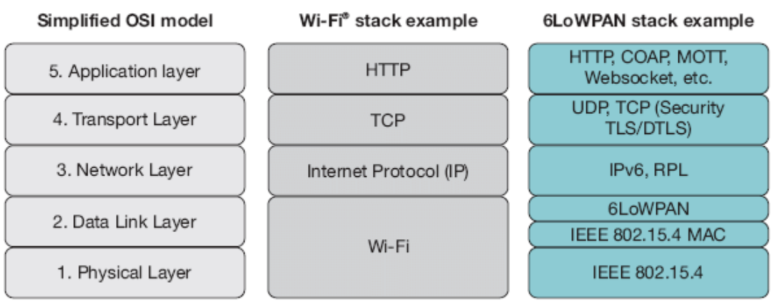
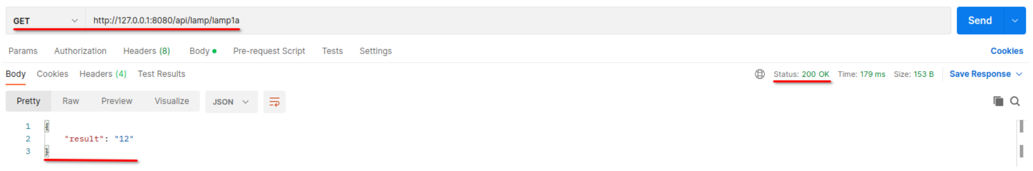


Figure 2.1: The OSI model, a Wi-Fi stack example and the 6LoWPAN stack exemple + ref

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).



http://127.0.0.1:8080/api/lamp/lamp1a

Status: 200 OK Time: 179 ms Size: 153 B

```
{
    "result": "12"
}
```

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

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.

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

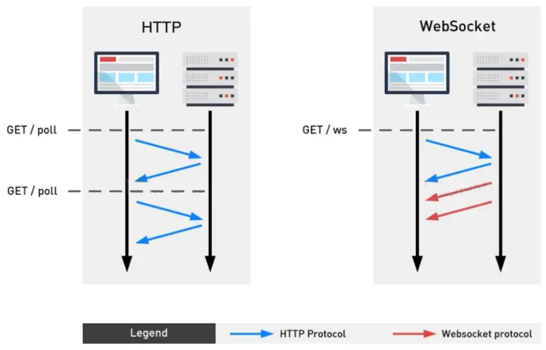


Figure 2.3: WebSockets allow the client-server connection to remain open, whereas traditionally the connection is closed after the server responds to a client request.

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.

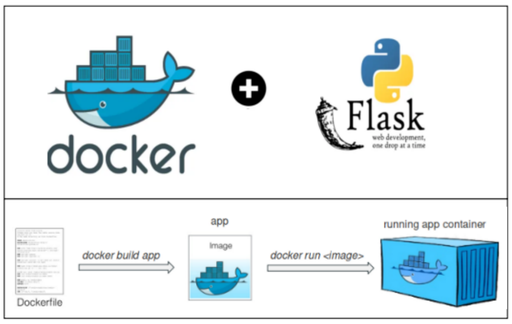


Figure 2.4: Containerization process with Docker

2.2 Front-end 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 Back-end

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

json vs xml, asynchrone pruts, hashing and salting, database modelling ORM, sql-injections, ajax calls

hashing, salting, rainbow tables, sqlinjection...

Hashing is an algorithm that calculates a fixed-size bit string value from a file. A file basically contains blocks of data. Hashing transforms this data into a far shorter fixed-length value or key which represents the original string. The hash value can be considered the distilled summary of everything within that file.

In cryptography, a salt is random data that is used as an additional input to a one-way function that hashes data, a password or passphrase.[1] Salts are used to safeguard passwords in storage.

Historically, only the output from an invocation of a cryptographic hash function on the password was stored on a system, but, over time, additional safeguards were developed to protect against duplicate or common passwords being identifiable (as their hashes are identical).[2] Salting is one such protection.

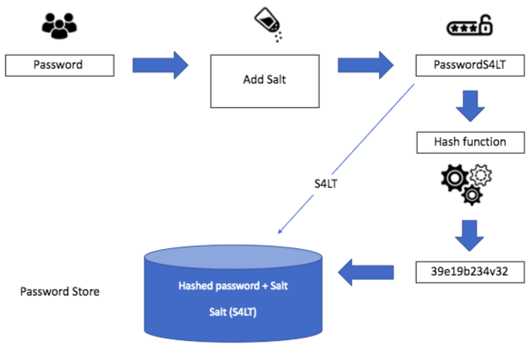


Figure 2.5: A secure way to store a password into a database through salting and hashing

Chapter 3

Methodology

write sections text

3.0.1 Project-architecture

add picture of the whole architecture

3.0.2 Front-end

add pictures of the UI

3.0.3 Back-end

Chapter 4

Results

write results

Chapter 5

Conclusions and recommendations

write conclusion and reco

recommendations: - git versioning (going back to previous versions and sharing code) - venv (python package conflicts)