UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

telecom BCN

Web authorization and authentication for single page applications (SPAs)

A Degree Thesis
Submitted to the Faculty of the
Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona
Universitat Politècnica de Catalunya
by

Enric Ruhi Velasco

In partial fulfilment
of the requirements for the degree in
TELEMATICS ENGINEERING

Advisor: José Luis Muñoz Tapia

Barcelona, May 2018

Abstract

The goal of this project is to study and implement security measures to protect against the most common attacks and to ensure the correct authorization and authentication of users in a single page application (SPA).

To do the project, we have studied the technologies used in practice for use authentication and authorization and the main attacks used to impersonate or steal users' credentials. Based on this knowledge, a theoretical scheme has been implemented for a secure user authorization and authentication that is resistant against the most attacks. Also, a proof of concept (PoC) of a SPA has been developed using the designed scheme.

Finally, we proved that the PoC is safe against the attacks contemplated in the project, however, we must keep in mind that this project is not contemplating major scale atacks (e.g. Hacked CA DigiNotar).

Resum

Aquest projecte té com a objectiu l'estudi i la implementació de mesures de seguretat per garantir la correcta autorització i autentificació d'usuaris en una pàgina web, concretament una "Single Page Application" (SPA), contra els atacs més habituals duts a terme contra una aplicació web.

Per dur a terme aquest projecte s'ha investigat les tecnologies i conceptes més utilitzats per realitzar l'autorització i autentificació d'usuaris i els principals atacs realitzats per suplantar la identitat o robar les credencials als usuaris. A partir d'aquests coneixements s'ha realitzat un esquema teòric per una segura autorització i autentificació d'usuaris contra els atacs estudiats, i s'ha fet una prova de concepte (PoC) d'una SPA seguint l'esquema realitzat per demostrar el seu correcte funcionament.

Finalment s'ha pogut observar que la PoC és segura contra els atacs contemplats en el projecte, tot i això s'ha de tenir en compte aquest projecte no estan contemplats atacs a gran escala com per exemple el pirateig de una CA com va passar amb DigiNotar.

Resumen

Este proyecto tiene como objetivo el estudio y la implementación de medidas de seguridad para garantizar una correcta autorización y autentificación de usuarios en una página web, concretamente una “Single Page Application” (SPA), contra los ataques más comunes realizados contra una aplicación web.

Para realizar el proyecto se ha investigado las tecnologías y conceptos más utilizados para realizar la autorización y autentificación de usuarios y los principales ataques realizados para suplantar la identidad o robar las credenciales de los usuarios. Partiendo de estos conocimientos se ha realizado un esquema teórico para una segura autorización y autentificación de usuarios contra los ataques estudiados y una prueba de concepto (PoC) de una SPA siguiendo este esquema para demostrar su correcto funcionamiento.

Finalmente se ha podido comprobar que la PoC es secura contra los ataques contemplados en el proyecto, sin embargo hay que tener en cuenta que este proyecto no ha tenido en cuenta ataques a gran escala como por ejemplo el pirateo de una CA como paso con DigiNotar.

Aquest treball està dedicat als meus pares i la meva germana, els quals han estat al meu costat en tot moment, durant tota la carrera, donant-me suport.

Moltes Gracies!

Acknowledgements

First and foremost, I would like to thank my thesis supervisor, José Luis Muñoz Tapia. Without his assistance and dedicated involvement in every step throughout the process, this project would have never been accomplished.

I would also like to show gratitude to my coworkers on everis, specially Jordi Magriña Cortes, Alex Perez Ujaque and Víctor Sauri Santacreu even though they were really busy they lend a helping hand when I asked for it.

Revision history and approval record

Revision	Date	Purpose
0	02/04/2018	Document creation
1	13/04/2018	Document revision
2	27/04/2018	Document revision
3	04/05/2018	Document revision
4	10/05/2018	Document revision

DOCUMENT DISTRIBUTION LIST

Name	e-mail
Enric Ruhi Velasco	enricruhi@gmail.com
José Luis Muñoz Tapia	lmunoz@entel.upc.edu

Written by:		Reviewed and approved by:	
Date	02/04/2018	Date	11/05/2018
Name	Enric Ruhi Velasco	Name	José Luis Muñoz Tapia
Position	Project Author	Position	Project Supervisor

Table of contents

List of Figures

List of Tables

Chapter 1

Introduction

Nowadays almost everyone uses internet daily, to be more specific, the world wide web (WWW). As this usage increases, we do more things online like the shopping, looking our bank account, socializing (Facebook, Instagram...) and many more. All this conveniences and uses come at the price of new risks. Without any online defense, we leave ourselves open to be a victim of fraud, theft and even property damage. Security is vital in keeping the web safe, everyone is a potential target, including governments and private corporations.

The objective of this project is to study and develop a secure way to authenticate and authorize the users logged in a web application, specifically a single page application (SPA).
To achieve this objective we will use the following concepts:

- Secure HTTP (HTTPS).
- Browser storage.
  - Cookies.
  - Session Storage.
  - Local Storage.
- Authentication Tokens.
  - Json Web Token (JWT).
  - Session ID
- Hashing algorithms.

We also studied the most commonly used attacks to steal the credentials or directly impersonate a user of a web application, such as cross-site scripting (XSS), cross-site request forgery (CSRF) and man in the middle (MITM).

After explaining the previous concepts, an overview of the MEAN stack is done. This stack is using MongoDB as database, NodeJS with Express as the back end server and Angular as the front end client. This application is used as a PoC (Proof of concept) to test the security measures developed.

This project is not a continuation of any other thesis or project. The work plan, milestons and time table can be found on the Appendix D.

Chapter 2

State of the art

2.1 Client side storage

2.1.1 What is client side storage?

When we talk about client side storage in the context of Web applications, we refer to the several ways of storing data at the client machine (where the browser is running). Currently, we have several ways of storing data in the client: cookies, local storage, session storage and IndexedDB. These storage systems have different features and capacities as shown in Table[1] 2.1.

	Cookies	Local Storage	Session Storage	IndexedDB
Capacity	4Kb	10Mb	10Mb	50% of free disk space
Browsers	HTML4/HTML5	HTML5	HTML5	HTML5
Accessible From	Any window	Any window	Same tab	Any window
Expires	Manually set	Never	On tab close	Never
Storage Location	Browser and server	Browser only	Browser only	Browser only
Sent with request	Yes	No	No	No

Table 2.1: Client side storage comparison.

Client storage is necessary for implementing features in Web apps like:

- Personalizing site preferences (e.g. showing a user's choice of custom widgets, color scheme, or font size).
- Persisting previous site activity (e.g. storing the contents of a shopping cart from a previous session, remembering if a user was previously logged in).
- Saving data and assets locally so a site will be quicker (and potentially less expensive) to download, or be usable without a network connection.
- Saving web application generated documents locally for use off-line.
- Recently, client storage is also being used to store (encrypted) sensitive data such as private keys of public key cryptography algorithms. This is used by Dapps (blockchain Web apps).

2.1.2 Cookies

Introduction

Traditionally, local storage has been used to overcome the fact that the HTTP protocol is stateless. This means that each HTTP request must contain enough information for creating the HTTP response. In other words, requests of a same application are

[1] We have to remark that the capacity and availability of these methods may vary depending on the browser type and version. In this regard, the storage capacity was tested on chrome, Version 65.0.3325.181 (Official Build, 64 bits).

not automatically associated with each other by HTTP. Thus, if the user selects some type of preference in a request, HTTP provides no means to link this fact in another request.

At first, to solve this problem, developers used cookies. Cookies are small documents, stored on the user's browser, that contain helpful information about users and their preferences (e.g. language, shopping cart, page layout...). For example if you selected to view a page in a certain language (e.g. English), the website would store that information on a cookie. This way, the next time you visited that website it would be able to read the cookie saved earlier and could "remember" the language without needing to select it again. What is saved on a cookie is up to the developer of the application. However, there are limitations on who can read the cookies and their size. The first limitation for security reasons is that only the same website that stores information to a cookie is able to access it. The second limitation for practical reasons imposes a maximum size of $\approx 4Kb$ for the cookie.

As the cookies gained popularity, more information was stored in them becoming a essential part of Web applications. Developers realized that the more information was stored on the cookies, they could better suit the needs of the users. However, since the cookie size is rather limited, developers came up with a workaround. The idea is to store the user preferences in the server not in the cookie and use a unique identifier for accessing these data. This identifier is typically called the session ID. So, when a client connects for the first time to a server, the server sends to the browser the session ID inside a cookie. Then, the browser stores the cookie and sends it each time the user browses the corresponding site. Finally, the server retrieves the state data for the specific client using the corresponding session ID.

Notice that the session ID is a way of storing an unlimited amount of user data because in fact, these user data is stored in the server which might have much more storage availability. This fact was a step towards third party cookies. As mentioned earlier, in general, only the same website that saved data to a cookie can access it later, but one website can contain bits from another. This pieces of website embedded on another website are able to access cookies that they previously stored on our computer. Usually many advertisements on multiple web pages are in fact from a single provider. This means that the advertisement provider can identify the websites that you browse and create a profile of you. Then, this is used to show specialized advertisements that you are more likely to click.

Notice that this creates privacy issues since you are profiled. This is a potential privacy concern that prompted European and U.S. lawmakers to take action in 2011 (Directive 2009/136/EC). European law requires that all websites targeting European Union member states gain "informed consent" from users before storing non-essential cookies on their device [1].

Cookie Syntax

When receiving an HTTP request, a server can send a `Set-Cookie` header with the response. The cookie is usually stored by the browser, and then the cookie is sent with requests made to the same server inside a Cookie HTTP header. An expiration date or duration can be specified, after which the cookie is no longer sent. Additionally, restrictions to a specific domain and path can be set, limiting where the cookie can be sent [2].

```
Set-Cookie: <cookie-name>=<cookie-value>
Set-Cookie: <cookie-name>=<cookie-value>; Expires=<date>
Set-Cookie: <cookie-name>=<cookie-value>; Max-Age=<non-zero-digit>
Set-Cookie: <cookie-name>=<cookie-value>; Domain=<domain-value>
Set-Cookie: <cookie-name>=<cookie-value>; Path=<path-value>
Set-Cookie: <cookie-name>=<cookie-value>; Secure
Set-Cookie: <cookie-name>=<cookie-value>; HttpOnly
Set-Cookie: <cookie-name>=<cookie-value>; SameSite=Strict
Set-Cookie: <cookie-name>=<cookie-value>; SameSite=Lax
```

Multiple directives are also possible, for example:

```
Set-Cookie: <cookie-name>=<cookie-value>; Domain=<domain-value>; Secure; HttpOnly
```

Depending on the duration of the cookie we have session cookies and persistent cookies (which can live across sessions).

HTTPOnly Flag

`HttpOnly` is an additional flag included in a `Set-Cookie` HTTP response header. Using the `HttpOnly` flag when generating a cookie helps mitigate the risk of client side script accessing the protected cookie. If the `HttpOnly` flag is included in the HTTP response header by the server, this means that the cookie cannot be accessed through a client side script. This

is enforced by the browser and limits the scope of the cookie only to HTTP requests. In particular, the attribute instructs the browser to omit the cookie when providing access to cookies via "non-HTTP" APIs.

If a browser does not support `HttpOnly` and a website attempts to set an `HttpOnly` cookie, the `HttpOnly` flag will be ignored by the browser, thus creating a traditional, script accessible cookie. As a result, the cookie (for example, including a session ID) becomes vulnerable to theft or modification by any malicious script [3].

Secure Flag

The `Secure` flag limits the scope of the cookie to "secure" channels. When a cookie has the `Secure` attribute, the browser will include the cookie in an HTTP request only if the request is transmitted over a secure channel, typically HTTP over Transport Layer Security (TLS) commonly known as (HTTPS) [4].

2.1.3 Web Storage

Web storage, sometimes known as DOM storage (Document Object Model storage) provides a way of storing persistent data similar to cookies with two relevant differences:

- Greatly enhanced capacity (around Megas).
- Data is not transferred in HTTP headers.

There are two main web storage types: local storage and session storage, behaving similarly to persistent cookies and session cookies respectively [5]:

- Session storage maintains a separate storage area for each given origin that's available for the duration of the page session (as long as the browser is open, including page reloads and restores).
- Local storage does the same thing as session storage, but persists even when the browser is closed and reopened.

Web storage is a key value storage in which values can only be strings. On the other hand, Web storage can be used together with server-side storage. For example, you could download from server a batch of music files used by a web game or music player app and store them inside a client-side database. The user would only have to download the music files once and on subsequent visits these music files would be retrieved from the client storage instead [6] of from the server. We can se an example of how to set, get and delete items from Web storage on the Figure: 2.1.

```
localStorage.setItem('myCat', 'Tom');
let cat = localStorage.getItem("myCat");
localStorage.removeItem("myCat");
```

Figure 2.1: Example of web storage basic functions.

2.1.4 IndexedDB

IndexedDB is a low-level API for client-side storage of significant amounts of structured data, including files/blobs. This API uses indexes to enable high-performance searches of data. While Web Storage is useful for storing smaller amounts of data, it is less useful for storing larger amounts of structured data [7]. We can see an example of an indexedDB creating the "Users" database with four fields: name, user, email and pass. Also how to add elements to said database on the Figure: 2.2.

```
let indexedDB = window.indexedDB || window.mozIndexedDB || window.webkitIndexedDB || window.msIndexedDB;
let dataBase = null;
startDB();

function startDB() {
  dataBase = indexedDB.open("objectDb", 1);
  dataBase.onupgradeneeded = function(e) {
    let active = dataBase.result;
    let objectDb = active.createObjectStore("users", {
      keyPath: 'id',
      autoIncrement: true
    });
    objectDb.createIndex('index_name', 'name', {
      unique: false
    });
    objectDb.createIndex('index_user', 'user', {
      unique: true
    });
    objectDb.createIndex('index_mail', 'email', {
      unique: true
    });
    objectDb.createIndex('index_pass', 'pass', {
      unique: false
    });
  };
  dataBase.onsuccess = function(e) {
    alert("Base de datos cargada correctamente");
    add();
  };

  dataBase.onerror = function(e) {
    alert("Error cargando la base de datos");
  };
}

function add() {
  let active = dataBase.result;
  let data = active.transaction(["users"], "readwrite");
  let objectDb = data.objectStore("users");
  let request = objectDb.put({
    name: "Enric",
    user: "eruhi",
    email: "randommail",
    pass: "password",
  });
  request.onerror = function(e) {
    alert(request.error.name + '\n\n' + request.error.message);
  };
}
```

Figure 2.2: Example of indexedDB set up.

The maximum browser storage space for IndexedDB is dynamic, it is based on your hard drive size. The global limit is calculated as 50% of free disk space.

IndexedDB is fairly new, and it is only supported by new releases of some of the major browsers (e.g. Firefox v58 released Jan 23 2018) for this reason it will not be used on this project.

2.2 Attacks

2.2.1 Most common threats

This project is focused on the most common threats on web authentication, most precisely, three of them. Cross-Site Request Forgery (CSRF), Cross-site Scripting (XSS) and Man In The Middle (MITM), this attacks are the most used either for the effectiveness of the thread, for example a successful MITM attack will be able to see all the communication (Including the login credentials). Another cause of a well spread attack is a low entrance barrier (difficulty) for example a CSRF attack is easy

to implement with basic web knowledge, but less effective than XSS and MITM. Finally XSS is always an active top threat to web security as we can see observing the top 10 security threat list from OWASP XSS was listed first on 2007, third on 2013 and seventh on 2017.

2.2.2 Cross-Site Request Forgery (CSRF)

CSRF is an attack that tricks the victim into submitting a malicious request. CSRF attacks target functionality that causes a state change on the server, such as changing the victim's email address or password, or purchasing something. With CSRF the attacker cannot retrieve data because the response is received by the victim not by the attacker. This is why CSRF attacks target state-changing requests.

This attack exploits the fact that the request inherits the identity and privileges of the victim to perform the undesired function on the victim's behalf. Regarding this, it results that for most sites, browser requests automatically include any credentials associated with the site, such as session cookie, IP address, browser windows domain credentials, and so forth.

Therefore, if the user is currently authenticated to the site, the site will have no way to distinguish between the forged request sent by the victim and a legitimate request sent by the victim.

It's sometimes possible to store the CSRF attack on the vulnerable site itself. Such vulnerabilities are called "stored CSRF flaws". This can be accomplished by simply storing an IMG or IFRAME tag in a field that accepts HTML, or by a more complex cross-site scripting attack. If the attacker can store a CSRF attack in the site, the severity of the attack is amplified. In particular, the likelihood is increased because the victim is more likely to view the page containing the attack than some random page on the Internet. The likelihood is also increased because the victim is sure to be authenticated to the site already [8].

2.2.3 Cross-site Scripting (XSS)

Cross-Site Scripting (XSS) attacks are a type of injection, in which malicious scripts are injected into otherwise benign and trusted web sites. XSS attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user. Flaws that allow these attacks to succeed are quite widespread and occur anywhere a web application uses input from a user within the output it generates without validating or encoding it.

An attacker can use XSS to send a malicious script to an unsuspecting user. The end user's browser has no way to know that the script should not be trusted, and will execute the script. Because it thinks the script came from a trusted source, the malicious script can access almost any cookies, session storage, local storage or other sensitive information retained by the browser and used with that site. These scripts can even rewrite the content of the HTML page.[9]

Cross-Site Scripting (XSS) attacks occur when:

- Data enters a Web application through an untrusted source, most frequently a web request.
- The data is included in dynamic content that is sent to a web user without being validated for malicious content.

The malicious content sent to the web browser often takes the form of a segment of JavaScript, but may also include HTML, Flash, or any other type of code that the browser may execute. The variety of attacks based on XSS is almost limitless, but they commonly include transmitting private data, like cookies or other session information, to the attacker, redirecting the victim to web content controlled by the attacker, or performing other malicious operations on the user's machine under the guise of the vulnerable site.

Types of Cross-Site Scripting

XSS attacks can generally be categorized into two categories: stored and reflected. There is a third, much less well known type of XSS attack called DOM Based XSS. [10]

DOM Based XSS (or as it is called in some texts, "type-0 XSS") is an XSS attack wherein the attack payload is executed as a result of modifying the DOM "environment" in the victim's browser used by the original client side script, so that the client side code runs in an "unexpected" manner. That is, the page itself (the HTTP response that is) does not change, but the client side code contained in the page executes differently due to the malicious modifications that have occurred in the DOM environment. [11]

This is in contrast to other XSS attacks (stored or reflected), wherein the attack payload is placed in the response page (due to a server side flaw).

Stored attacks are those where the injected script is permanently stored on the target servers, such as in a database, in a message forum, visitor log, comment field, etc. The victim then retrieves the malicious script from the server when it requests the stored information. Stored XSS is also sometimes referred to as Persistent or Type-I XSS.

An example of this type of attack, is if a forum or a blog have not the correct input and output validations and escaping manage to save to the database a script as a message of the page, this way any user that loads that message will be victim of the script that was loaded there, we can see it on the Figure: 2.3 .

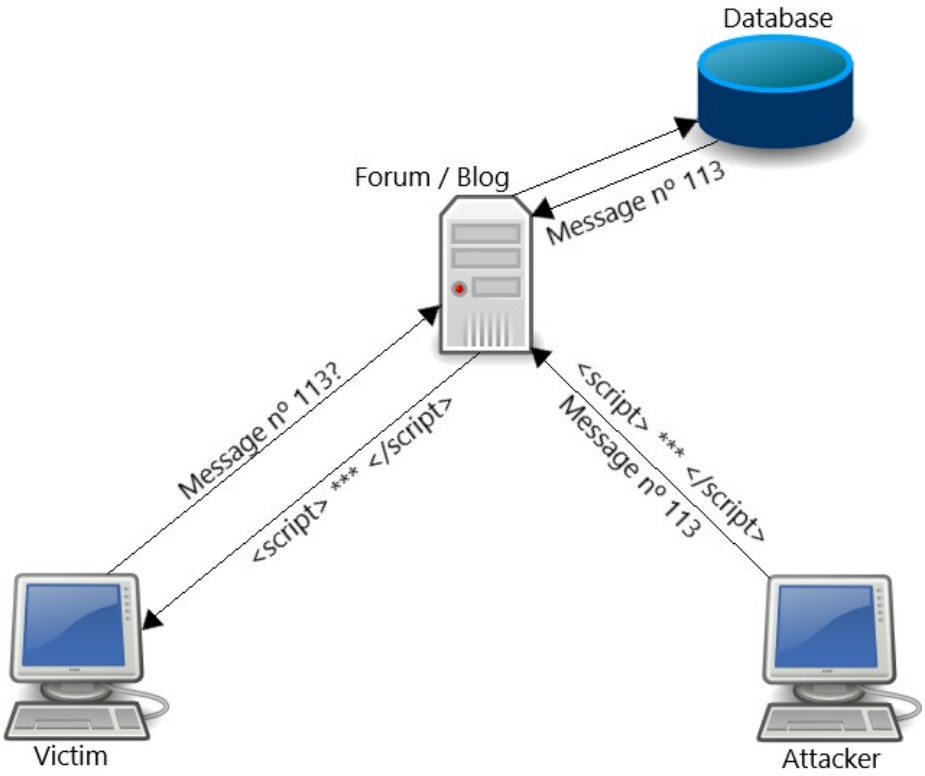


Figure 2.3: Example of Stored XSS.

Reflected attacks are those where the injected script is reflected off the web server, such as in an error message, search result, or any other response that includes some or all of the input sent to the server as part of the request. Reflected attacks are delivered to victims via another route, such as in an e-mail message, or on some other web site. When a user is tricked into clicking on a malicious link, submitting a specially crafted form, or even just browsing to a malicious site, the injected code travels to the vulnerable web site, which reflects the attack back to the user's browser. The browser then executes the code because it came from a "trusted" server. Reflected XSS is also sometimes referred to as Non-Persistent or Type-II XSS, we can see how an attacker might exploit reflected XSS on the Figure:2.4[12].

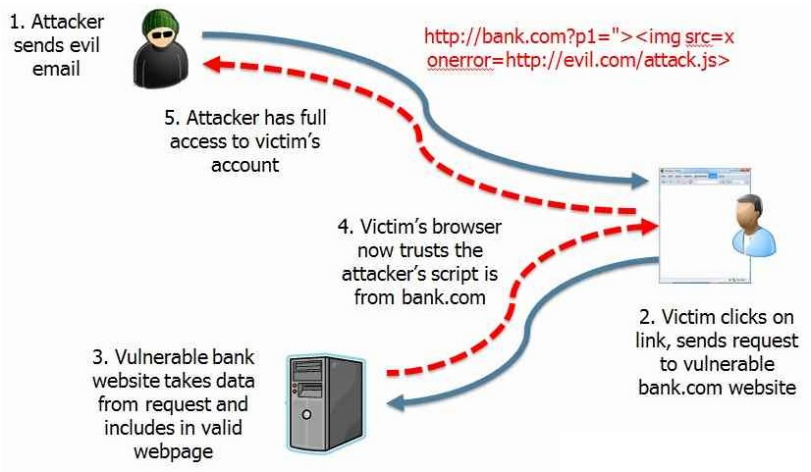


Figure 2.4: Example of Reflected XSS.

Since this attacks overlap, you can have both Stored and Reflected DOM Based XSS. You can also have Stored and Reflected Non-DOM Based XSS too, but that's confusing, so to help clarify things, starting about mid 2012, the research community proposed and started using two new terms to help organize the types of XSS that can occur:

- Server XSS
- Client XSS

Server XSS occurs when untrusted user supplied data is included in an HTML response generated by the server. The source of this data could be from the request, or from a stored location. As such, you can have both Reflected Server XSS and Stored Server XSS.

In this case, the entire vulnerability is in server-side code, and the browser is simply rendering the response and executing any valid script embedded in it.

Client XSS occurs when untrusted user supplied data is used to update the DOM with an unsafe JavaScript call. A JavaScript call is considered unsafe if it can be used to introduce valid JavaScript into the DOM. This source of this data could be from the DOM, or it could have been sent by the server (via an AJAX call, or a page load). The ultimate source of the data could have been from a request, or from a stored location on the client or the server. As such, you can have both Reflected Client XSS and Stored Client XSS.

With these new definitions, the definition of DOM Based XSS doesn't change. DOM Based XSS is simply a subset of Client XSS, where the source of the data is somewhere in the DOM, rather than from the Server.

Given that both Server XSS and Client XSS can be Stored or Reflected, this new terminology results in a simple, clean, 2 x 2 matrix with Client & Server XSS on one axis, and Stored and Reflected XSS on the other axis as shown in the Figure: 2.2 where the top row indicates where the untrusted data is used and the first column is the data persistence.

XSS	Server	Client
Stored	Stored Server XSS	Stored Client XSS
Reflected XSS	Reflected Server XSS	Reflected Client XSS

Table 2.2: Merge of the two XSS organisations

Note that DOM based XSS is a subset of client XSS were the data source is from the DOM only.

Stored vs Reflected only affects the likelihood of successful attack, not the nature of vulnerability or the most effective defence.

2.2.4 Angular's cross-site scripting security model

To systematically block XSS bugs, Angular treats all values as untrusted by default. When a value is inserted into the DOM from a template, via property, attribute, style, class binding, or interpolation, Angular sanitizes and escapes untrusted values.

Angular templates are the same as executable code: HTML, attributes, and binding expressions (but not the values bound) in templates are trusted to be safe. This means that applications must prevent values that an attacker can control from ever making it into the source code of a template. Never generate template source code by concatenating user input and templates. To prevent these vulnerabilities, we should use the offline template compiler, also known as template injection. [13]

Sanitization and security contexts

Sanitization is the inspection of an untrusted value, turning it into a value that's safe to insert into the DOM. In many cases, sanitization doesn't change a value at all. Sanitization depends on context: a value that's harmless in CSS is potentially dangerous in a URL. Angular defines the following security contexts:

- HTML is used when interpreting a value as HTML, for example, when binding to innerHtml.
- Style is used when binding CSS into the style property.
- URL is used for URL properties, such as <a href>.
- Resource URL is a URL that will be loaded and executed as code, for example, in <script src>.

Angular sanitizes untrusted values for HTML, styles, and URLs; sanitizing resource URLs isn't possible because they contain arbitrary code. In development mode, Angular prints a console warning when it has to change a value during sanitization.

Sanitization example

The following template binds the value of htmlSnippet, once by interpolating it into an element's content, and once by binding it to the innerHTML property of an element:

```
<h3>Binding innerHTML</h3>
<p>Bound value:</p>
<p class="e2e-inner-html-interpolated">{{htmlSnippet}}</p>
<p>Result of binding to innerHTML:</p>
<p class="e2e-inner-html-bound" [innerHTML]="htmlSnippet"></p>
```

Interpolated content is always escaped—the HTML isn't interpreted and the browser displays angle brackets in the element's text content.

For the HTML to be interpreted, bind it to an HTML property such as innerHTML. But binding a value that an attacker might control into innerHTML normally causes an XSS vulnerability. For example, code contained in a <script> tag is executed:

```
htmlSnippet = 'Template <script>alert("0wned")</script> <b>Syntax</b>';
```

Angular recognizes the value as unsafe and automatically sanitizes it, which removes the <script> tag but keeps safe content such as the text content of the <script> tag and the <b> element. We can see the result in the Figure:2.5 .

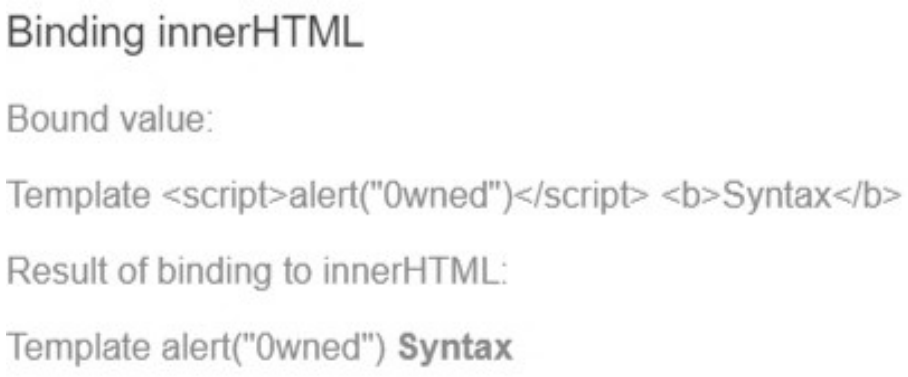


Figure 2.5: Example of Angular sanitation.

2.2.5 Man In The Middle (MITM)

The man-in-the middle attack intercepts a communication between two systems. For example, in an http transaction the target is the TCP connection between client and server. Using different techniques, the attacker splits the original TCP connection into 2 new connections, one between the client and the attacker and the other between the attacker and the server. Once the TCP connection is intercepted, the attacker acts as a proxy, being able to read, insert and modify the data in the intercepted communication.

The MITM attack is very effective because of the nature of the http protocol and data transfer which are all ASCII based. In this way, it's possible to view and interview within the http protocol and also in the data transferred. So, for example, it's possible to capture a session cookie reading the http header, but it's also possible to change an amount of money transaction inside the application context.

The MITM attack could also be done over an https connection by using the same technique; the only difference consists in the establishment of two independent SSL sessions, one over each TCP connection. The browser sets a SSL connection with the attacker, and the attacker establishes another SSL connection with the web server. In general the browser warns the user that the digital certificate used is not valid, but the user may ignore the warning because he doesn't understand the threat. In some specific contexts it's possible that the warning doesn't appear, as for example, when the Server certificate is compromised by the attacker or when the attacker certificate is signed by a trusted CA and the CN is the same of the original web site.

MITM is not only an attack technique, but is also usually used during the development step of a web application or is still used for Web Vulnerability assessments [14]. We can see examples of MITM on the section 2.3.2.

2.3 REST Web Services

2.3.1 HTTP

Hypertext Transfer Protocol (HTTP) is an application-layer protocol for transmitting hypermedia documents, such as HTML. It was designed for communication between web browsers and web servers, but it can also be used for other purposes. HTTP follows a classical client-server model, with a client opening a connection to make a request, then waiting until it receives a response. HTTP is a stateless protocol, meaning that the server does not keep any data (state) between two requests. Though often based on a TCP/IP layer, it can be used on any reliable transport layer; that is, a protocol that does not lose messages silently, such as UDP.[15]

2.3.2 HTTPS

Hyper Text Transfer Protocol Secure (HTTPS) is the secure version of HTTP, the protocol over which data is sent between your browser and the website that you are connected to. The 'S' at the end of HTTPS stands for 'Secure'. It means all communications between your browser and the website are encrypted.[16]

How does HTTPS work?

HTTPS pages typically use one of two secure protocols to encrypt communications - SSL (Secure Sockets Layer) or TLS (Transport Layer Security). Both the TLS and SSL protocols use what is known as an 'asymmetric' Public Key Infrastructure (PKI) system. An asymmetric system uses two 'keys' to encrypt communications, a 'public' key and a 'private' key. Anything encrypted with the public key can only be decrypted by the private key and vice-versa. This is illustrated by the Figure: 2.6.

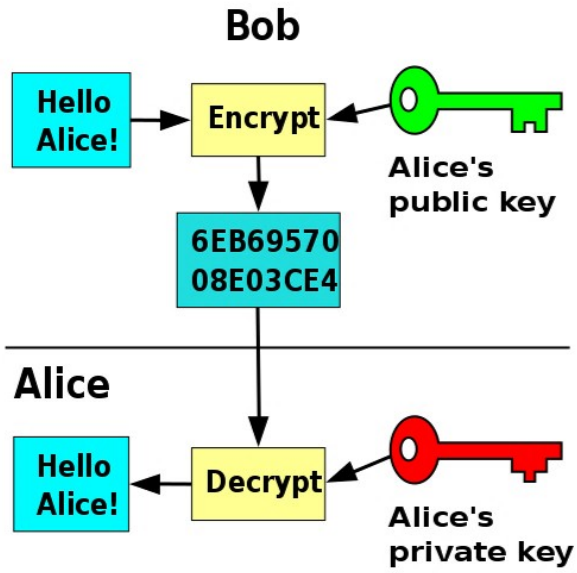


Figure 2.6: Asymmetrical cryptography work flow.

What is a HTTPS certificate?

When you request a HTTPS connection to a web page, the website will initially send its SSL certificate to your browser. This certificate contains the public key needed to begin the secure session. Based on this initial exchange, your browser and the website then initiate the 'SSL handshake'. The SSL handshake involves the generation of shared secrets to establish a uniquely secure connection between yourself and the website. The greatest benefit of HTTPS over HTTPS is that your information is protected and not accessible for anyone to see. This gains importance when the information is more sensible, credit card number, passwords, conversations you want to keep private, etc.

When a trusted SSL Digital Certificate is used during a HTTPS connection, users will see a padlock icon in the browser address bar. When an Extended Validation Certificate is installed on a web site, the address bar will turn green. We can see the example of the UPC university front page (https://www.upc.edu/ca) on chrome browser on the Figure: 2.7.

Figure 2.7: Padlock icon indicating trusted HTTPS certificates.

Why are HTTPS and certificates so important?

The implementation HTTPS (HTTP over TLS/SSL) is a key factor to protect the privacy and security of the users, any communication over plain HTTP can be read by any one intercepting at the communication between the start and the destiny of the said message (man in the middle). To do that the attacker will trick the network to believe that his machine is now the router, this way all the traffic will go through his PC and then he will forward it to the correct location after he took a good look, saved the information, etc. This is terrible since if there is any sensible information such as passwords it would be send in plain text. To illustrate the process we can look at the Figure: 2.8

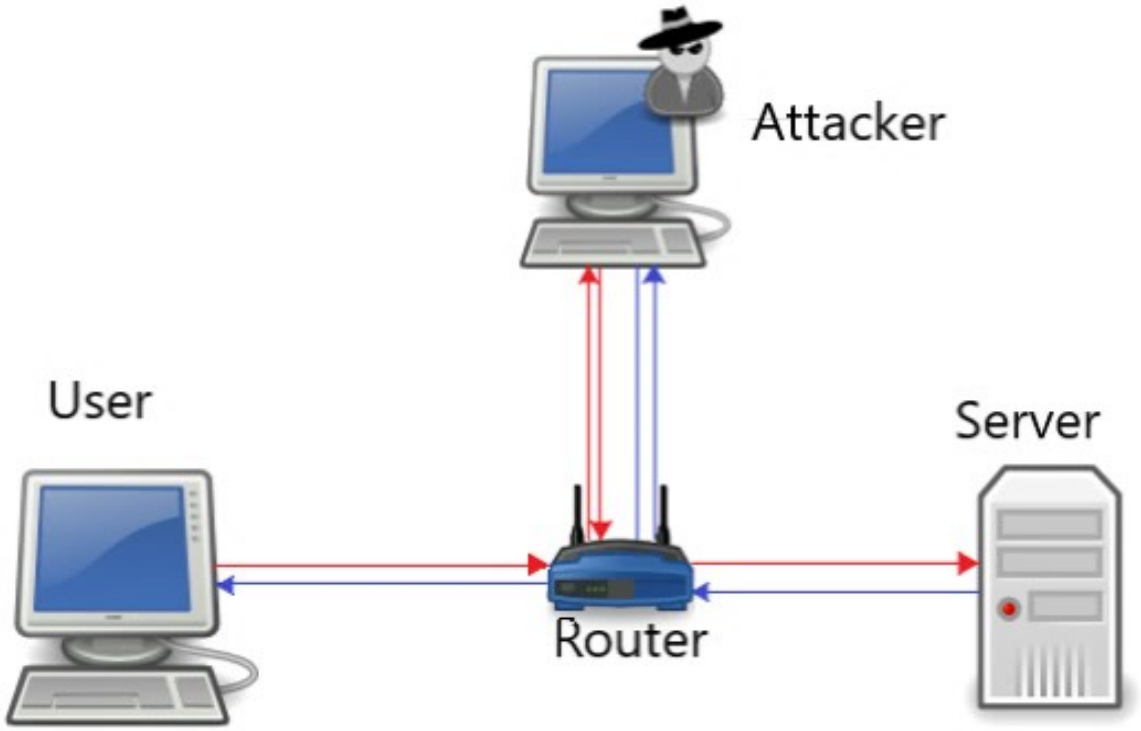


Figure 2.8: Traffic send over HTTP being intercepted by an attacker.

The next step to prevent that is to send the information over HTTPS but doing that only moves the problem back a stage, since the communication will be using asymmetrical cryptography that consist of an exchange of keys, simplifying the server will send a public key to encrypt the message to the user and the only key to unlock it would be the private key, the one only the server has access. To illustrate the process we can look at the Figure:

Yes, this solves the initial problem of all the messages going on plain text, and now if someone is sitting in the middle it will only see encrypted that that will make no sense. But all we really done we moved the problem back a stage, since the first stage on exchanging keys that has to go on plain text and someone in the middle can change that and send his own public key to the user keeping the public key of the server to impersonate the actual user. The process to do that is the following illustrated by the Figure: 2.9 .

- Attacker takes the public key from the server.
- Sends his own public key to the user.
- User will encrypt the message with the attacker public key and send it.
- Attacker will intercept it, decrypt it and read it.
- Encrypt it wiht the server's public key and send it to the server.

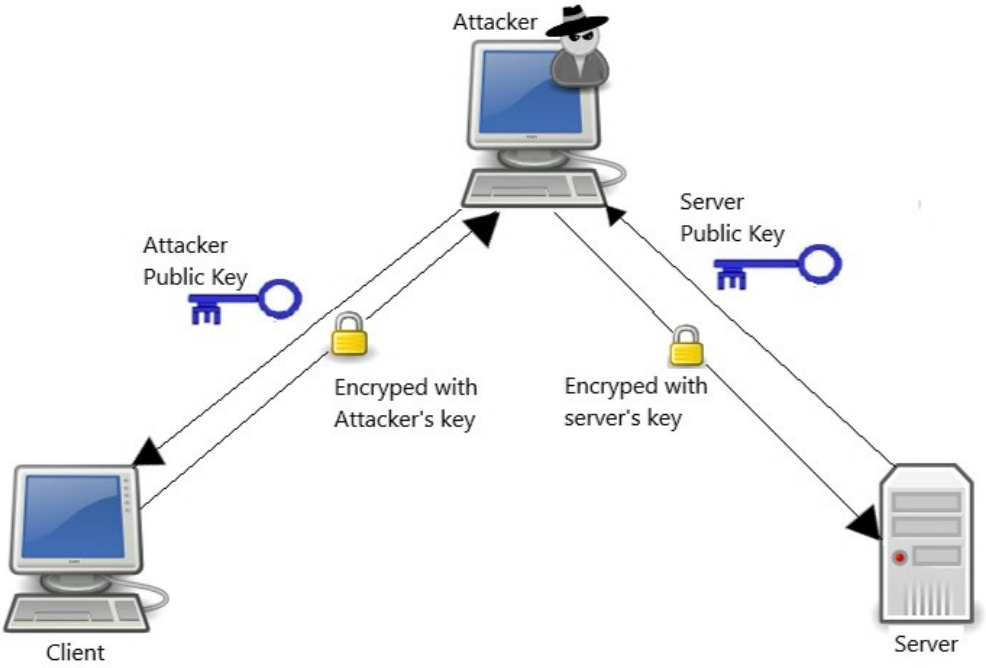


Figure 2.9: Man in the middle impersonating the server.

The solution to this problem is using signed certificates by a certificate authority (CA). A certificate authority is a third party vouching for the set of public keys exchanged. This way if the attacker tries to change the keys it will be noticed, the user will be alerted, since the keys will not be signed by the CA. To illustrate this process we can look at the Figure:2.10.

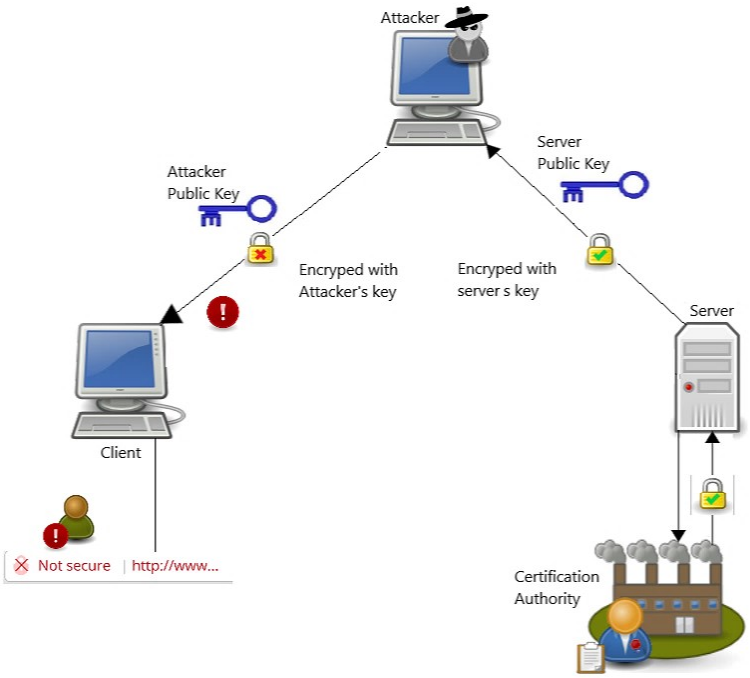


Figure 2.10: Man in the middle key change noticed by the client.

Now the problem is to know what certificate authorities to trust. This is when the end users need to trust the companies such as Apple, Microsoft, Google, etc. Since the trusted certificates are pre-installed on your device or browser. Now we the attacker is on lock because the full circle of "trust" placed.

2.3.3 Introduction to APIs

What is an API?

API stands for " Application Programming Interface ". An API is a set of subroutine definitions, protocols, and tools for building application software. In general terms, it is a set of clearly defined methods of communication between various software components. It provides developers with standard commands for performing common operations so they do not have to write the code from scratch. An API may be for a web-based system, operating system, database system, computer hardware or software library.

Purpose of an API

Just as a graphical user interface makes it easier for people to use programs, application programming interfaces make it easier for developers to use certain technologies in building applications. By abstracting the underlying implementation and only exposing objects or actions the developer needs, an API simplifies programming. While a graphical interface for an email client might provide a user with a button that performs all the steps for fetching and highlighting new emails, an API for file input/output might give the developer a function that copies a file from one location to another without requiring that the developer understand the file system operations occurring behind the scenes.[17]

REST APIs

REST stands for Representational State Transfer. A REST API defines a set of functions which developers can perform requests and receive responses via hypertext transfer protocol (HTTP) or Secure Hyper Text Transfer Protocol (HTTPS) such as GET and POST.

2.4 API Authentication

2.4.1 Session ID

A session ID is a unique number that a Web site's server assigns a specific user for the duration of that user's visit (session). The session ID can be stored as a cookie, form field, or URL (Uniform Resource Locator). Some Web servers generate session IDs by simply incrementing static numbers. However, most servers use algorithms that involve more complex methods, such as factoring in the date and time of the visit along with other variables defined by the server administrator.

Session ID is a token by reference, that means that we will need a third party storing service (Memory, database, etc) to know if it is valid or not.

2.4.2 JSON Web Token (JWT)

JSON Web Token (JWT) is an open standard [18] that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the HMAC algorithm), a public/private key pair using RSA and many others.

There is actually three parts on a JWT, first we have the header, then the payload and finally the signature all of them encoded on base64URL. If you want to see how a JWT looks like this link will provide with a dynamic one.[19]

The specifications tell us a lot of information we can put on the payload, called claims, we use the signature to verify if they are true or false. Some claims are reserved like "iss" (issuer), "sub" (subject), "exp" (expiration time) and more, to know more about JWT go to the standard referenced above (RFC 7517).

JWT is a token by value, that means that we can verify from the server if it's valid or forged.

What is JWT and used for?

JSON web tokens are used, most commonly, for authentication. Once the user is logged in, each subsequent request will include the JWT, allowing the user to access routes, services, and resources that are permitted with that token. It is used to exchange information, because JWTs can be signed for example, using public/private key parts, we can be sure the senders are who they say they are. Additionally, as the signature is calculated using the header and the payload, we can also verify that the content hasn't been tampered with.[20]

2.4.3 Web Authentication

Browser without authentication

First of all we will see an example of the most basic communication, without any authentication.

When a user sings in on a website it enters the user and password. Then the browser sends both to the server, the server asks the data base if the user exist, if so, the data base will return the hashed and salted information. After the server would do the needed security protocol to check the password. If everything is ok, the server can finally send the response and that is when the browser stores the key/value pair, on the example NAME = ALICE, the cookie. When the user navigates on the website the cookie is send to the server automatically. Finally the server returns a dedicated page for the user "HELLO ALICE". Figure 2.11 illustrates the process explained above.

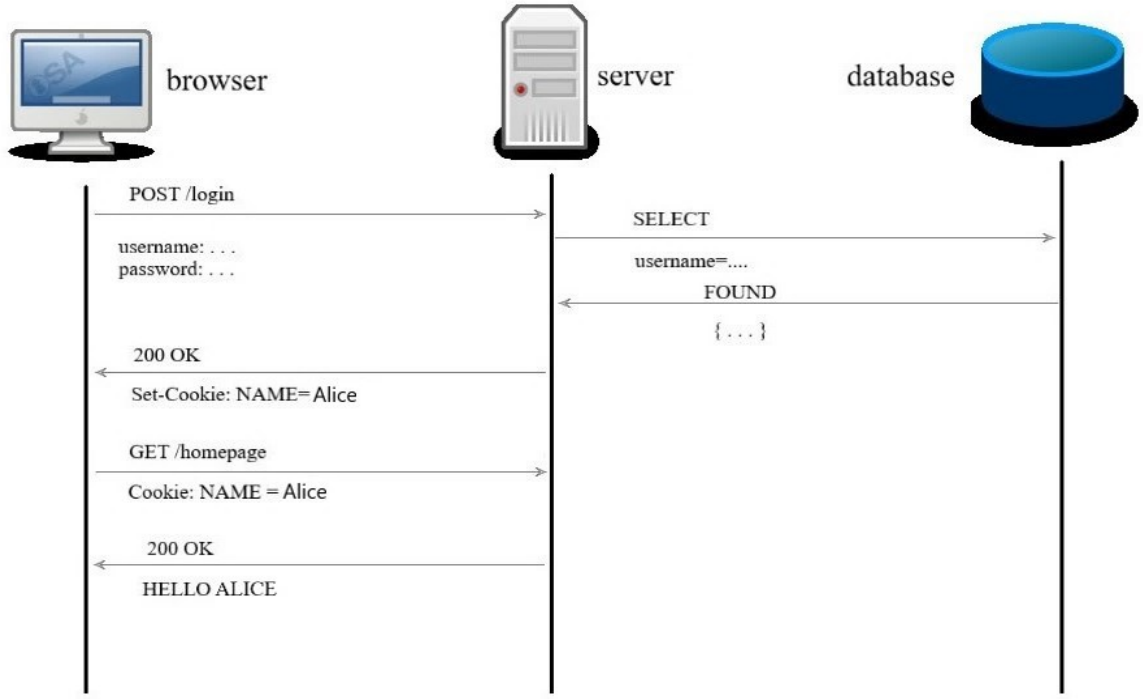


Figure 2.11: Work flow without authentication

In this case the server is easily tricked as we can see on the Figure 2.12, we can say to the server we are Bob when we are actually Alice.

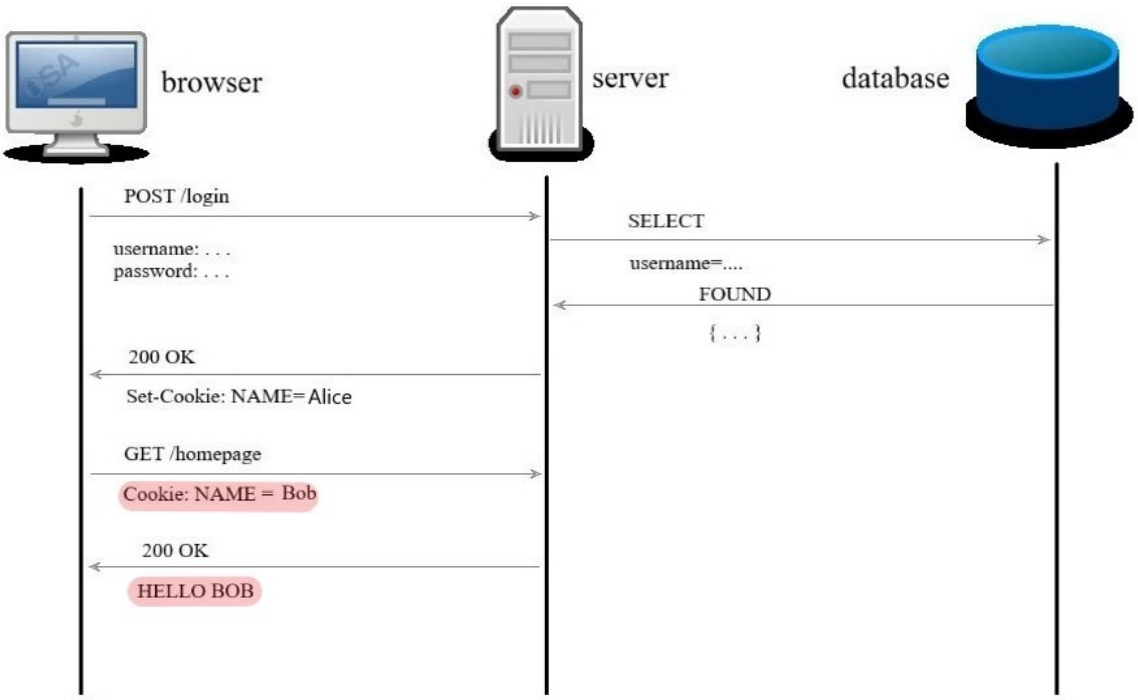


Figure 2.12: Server tricked by the browser

We now see the importance of the authentication. The server needs a way to trust what the browser is giving is correct.

Browser with Session ID authentication

This example solves the trust issue of the server explained on the example above using session ID, an authentication token by reference.

Following the steps of the previous example, once the server is about to respond, it generates a random ID for example A42, and using a third party storing service (memory, database ...) to store and relating it to all the information the server has about Alice. Then the server answer the browser with the ID instead of the NAME. The browser, as we saw before, sends the cookies to the server. This time, the server will need the third party storing service (Memory, database, etc) to transform this ID to the information of Alice, so doing a round trip to the memory, it knows it was Alice.

Figure 2.13 illustrates the process explained above.

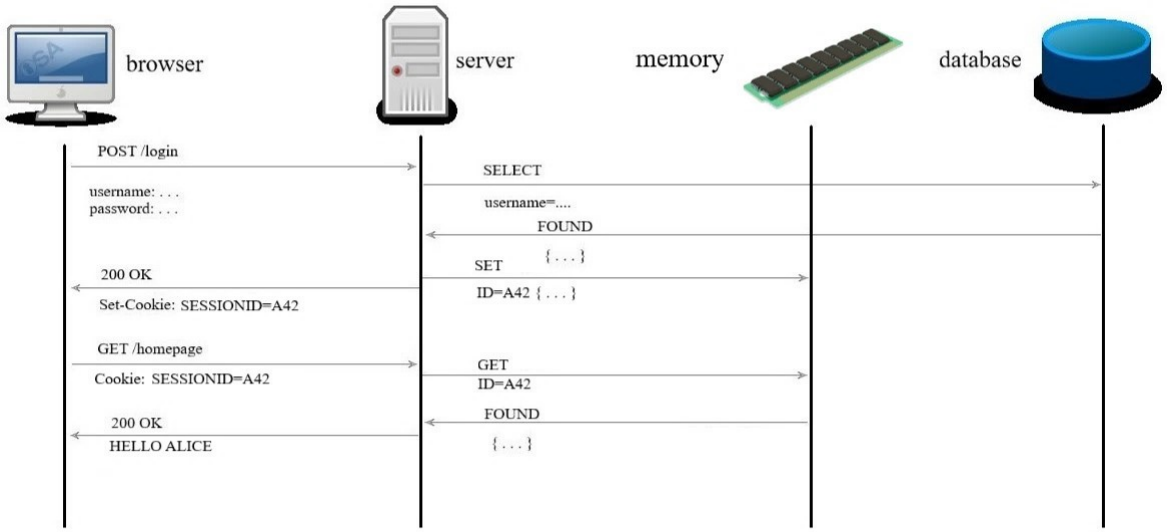


Figure 2.13: Work flow with SessionID

Problems with Session ID

There are several problems with session IDs. If the site is successful, one server might not be enough, we could just put two servers and a load balancer, but if the site is big enough, we will have a lot of servers, each with its own memory. With a load balancer, we could end at the server that has the information we need or on one that knows nothing about it as illustrated by the Figure:2.14.

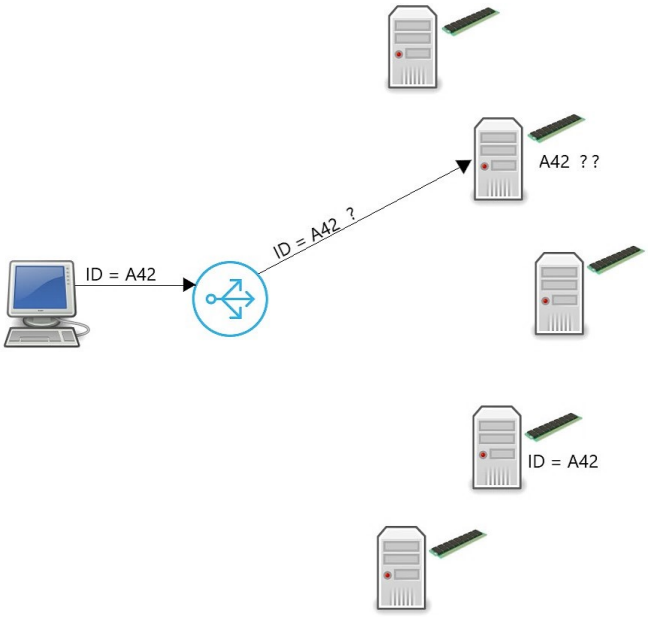
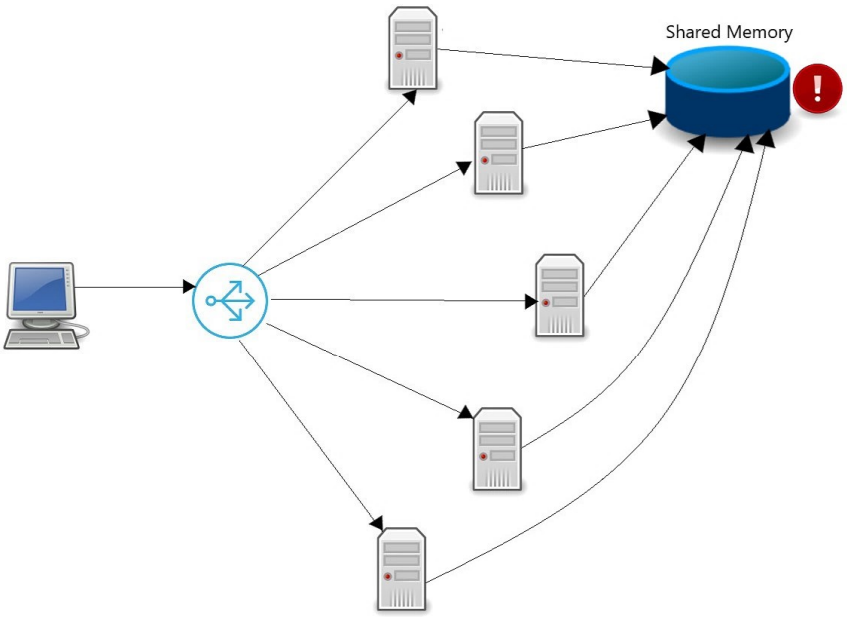


Figure 2.14: Looking for the ID on the wrong server.

An easy solution would be to make a shared memory among all the servers, they would ask there for the information, but now we have a single point of failure and if it fails all the system fails as we can see on the Figure:2.15.



s

Figure 2.15: Shared memory (Single point of failure).

We could use a distributed memory so every node knows about every session, if a node fails some users might be disconnected but the site will not be down.

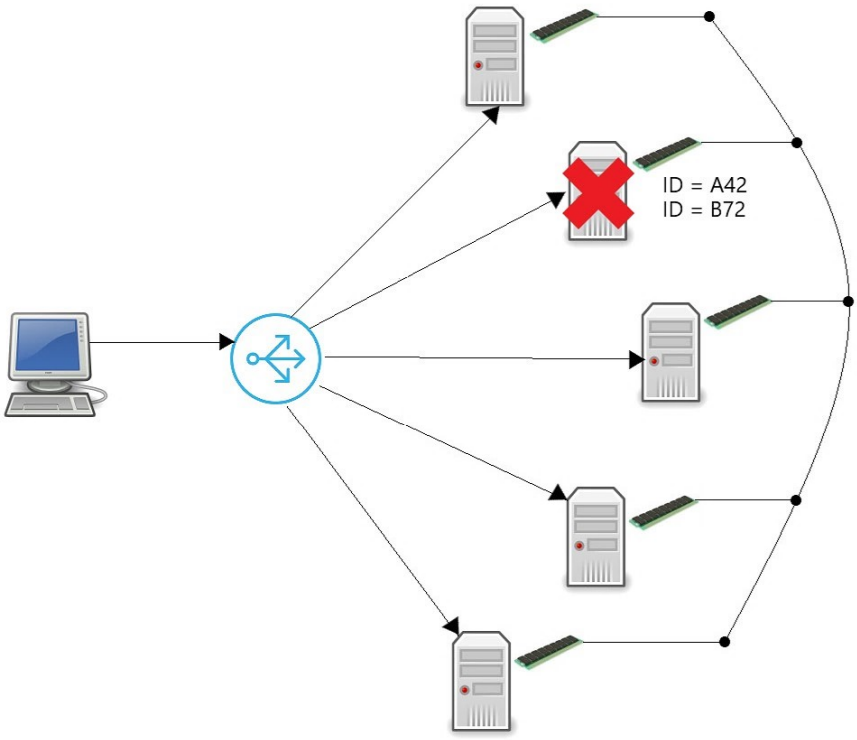


Figure 2.16: Distributed memory for al servers.

As we can see on the Figure: 2.16 if the node with the session 42 and 72 goes down, this two users will not be able to access but the rest of the site will keep working.

As we see from this examples, no solution is perfect, but if we managed the data on the client side it could be simpler.

Browser with JWT authentication

We have another solution, JSON web token (JWT), that is a token by value. This approach starts as the other two examples, we start with a round trip to the database and check the password. With the session ID we need a third party service, but now, we do not. Now we just need to write and sign the information. After that the server send the data to the browser like the first example the browser stores the information on the cookies and then sends them to the server. This time the server will verify if the information is trustworthy and then read it so we won't have the same problem as the example one.

Figure 2.17 illustrates the process explained above.

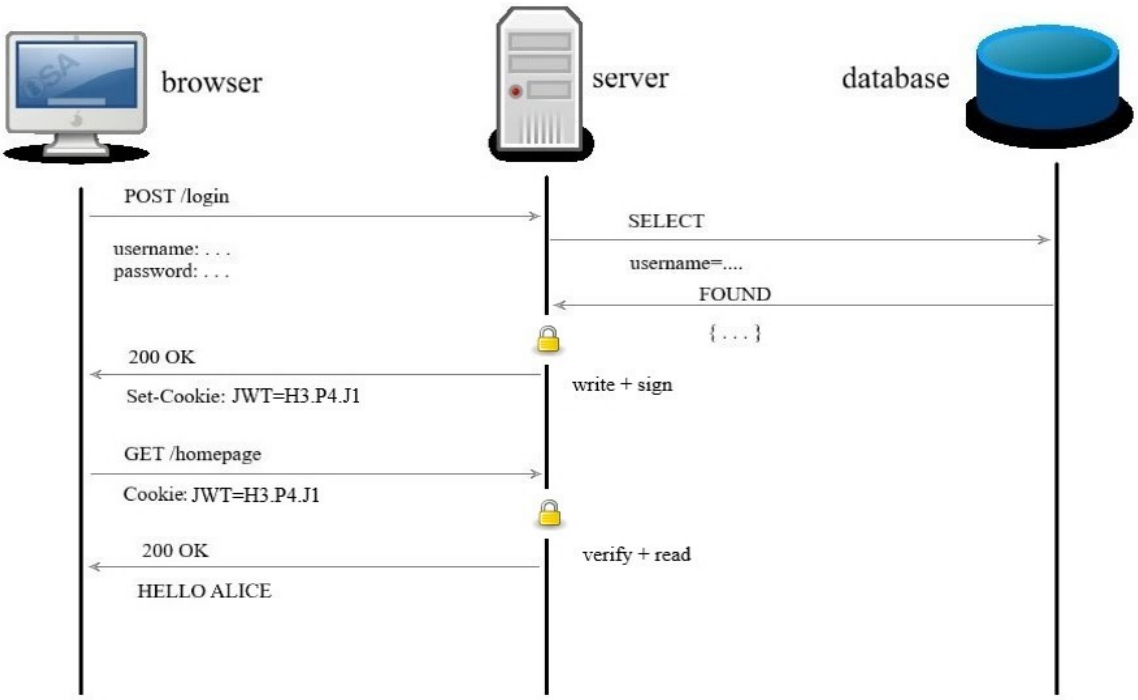


Figure 2.17: Work flow with JWT

2.4.4 JWT pros and cons

Benefits of JWT

The first benefit is the load balancing, if we look at the problems with session ID we do not need a shared or distributed memory, we just need some CPU to calculate a base64 and parse a JSON.

Second since it is a standard it works with any kind of language, you could have a multi-language architecture using the same tokens.

Finally it can be a key part of micro services since it is stateless so we do not need big database to store all micro service data and we avoid again the single point of failure.

Drawbacks of JWT

The first and the most important drawback is the token revocation, what can we do when we have to revoke one token but we won't want to revoke all of them. The problem is, if we have our architecture how we manage to revoke one and not the other. The only way is to use a blacklist to store all the JWT ids that are not suposed to pass. It is a nice solution, but in security it is preferred to have the opposite case, a list that are able to access the system. It is preferred because if the blacklist fails, we are letting every one access. The blacklist is easier to handle but in a secure point of view is problematic.

Second drawback is the security of single page applications, it is not that much of a drawback than a need for a developer to be careful. If we use cookies or local storage to store this JWT we can be subjected to various attacks. On one hand we have cross-site scripting (XSS), since we have injection content on our site, if there are scripts injected on the page, they can read the local storage and some of the cookies, this makes storing the token in local or session storage insecure. On the other hand, if we store our JWT on a cookie, if not well protected can be subject to XSS too, but most important can be easily attacked using cross site request forgery (CSRF). Finally all non-secured HTTP connections, a public WI-FI for example, all the information is clear.

2.5 CORS

2.5.1 What is CORS?

CORS stands for Cross-Origin Resource Sharing. It allows you to break the same origin policy (SOP) of a browser. The same-origin policy restricts how a document or script loaded from one origin can interact with a resource from another origin. It is a critical security mechanism for isolating potentially malicious documents. [21]

For a better understanding of both CORS and SOP we will provide with a simple example. If we access to the page "moo.com", this page comes with some HTML, CSS and JS. When the JS of the page runs, it might need some data from a server, in this case from "foo.com" that contains an API with the necessary data. When the request is made and "foo.com" response. Since we are requesting data from "foo.com" and not the initial "moo.com" this data is blocked by the browser and it does not let the data propagate to the page, the Figure: 2.18 illustrates this process.

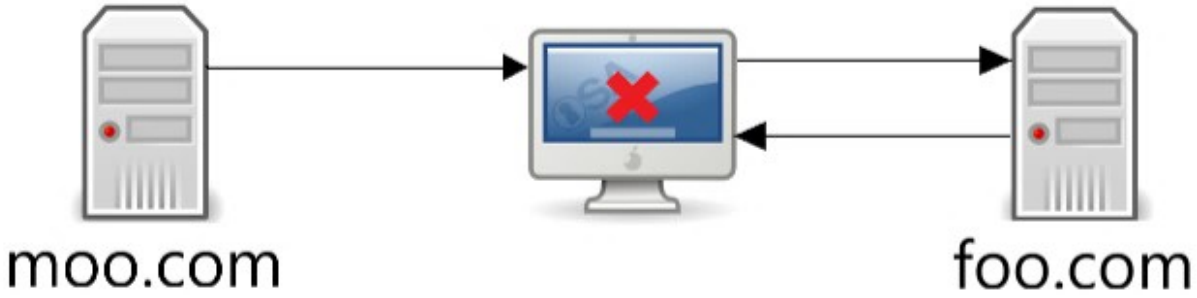


Figure 2.18: Browser blocking response from "foo.com"

The reason this exist is to prevent unwanted access of someone's API. If we want to share our API with others, this is were CORS comes in play. If "foo.com" and "moo.com" were owned by the same person, he can set up CORS so "moo.com" can request data from "foo.com".

2.5.2 How does CORS work?

To unlock certain domains to obey SOP. On one hand we need a header called "origin" on the request to identify who is requesting the information. On the other hand we need a header called "Access-Control-Allow-Origin" to notify the browser the origins we allow. This header must be the same of the "origin" header or "*" (That means all origins are acceped) to prevent the blocking of the response. This process is illustrated on the Figure:2.19.

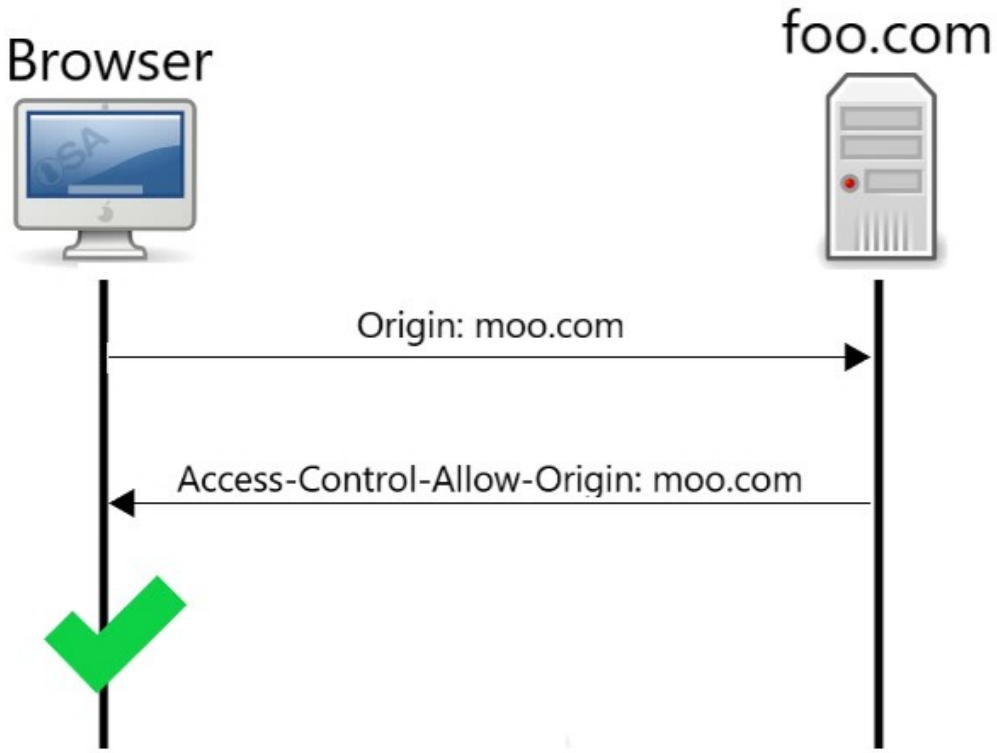


Figure 2.19: GET request allowed example.

This example is an allowed response, as said before if the response header was not "moo.com" or "*" the browser will block the response. An important point is that what is blocked is the response, not the request, but the response this is why this example is ok with a GET request because it will not change anything on the server side, but if you issue a PUT, DELETE or any other request that changes the state of the data, it goes to "foo.com" this does any process he needs and then the response is blocked the server state is already changed. CORS gets around this sending what it is called a preflight request. A preflight request is a HTTP request called options that will include the previous "origin" header and another called "Access-Control-Request-Method" this header will contain the method that eventually is wanted to send the server. The server will response with the previous "Access-Control-Allow-Origin" and another one called "Access-Control-Allow-Methods" that can contain one or more methods that are allowed. If the method on the request is inside the list of the response the same process as before will take place. An example of this process is illustrated on the Figure: 2.20.

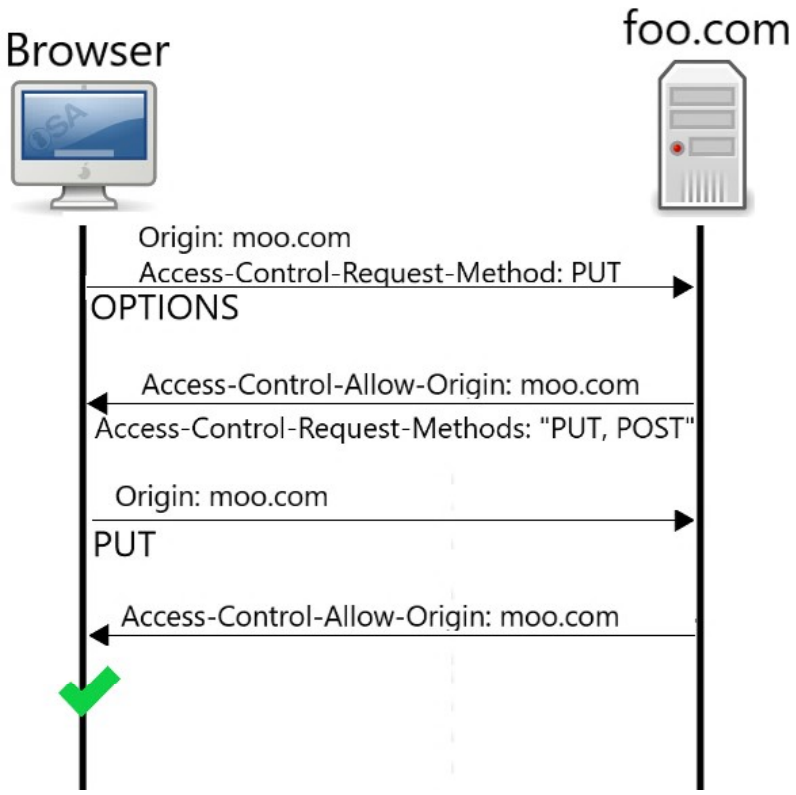


Figure 2.20: Options header example.

Since on our example both origin and PUT request are the allowed by CORS the response will not be blocked, if either the origin on any of the both communications is different (Not moo.com or *) or the PUT method is not on the list of allowed methods our request will respectively not be send or be blocked.

Some requests don't trigger a CORS preflight. Those are called "simple requests". A so called "simple request" is one that meets all the following conditions:

- The only allowed methods are:
  - GET
  - HEAD
  - POST
- The only headers which are allowed to be manually set are:
  - Accept
  - Accept-Language
  - Content-Language
  - Content-Type (but note the additional requirements below)
  - Last-Event-ID
  - DPR
  - Save-Data
  - Viewport-Width
  - Width
- The only allowed values for the Content-Type header are:

- application/x-www-form-urlencoded
- multipart/form-data
- text/plain

2.5.3 Experimenting with CORS

There are multiple on-line tools to test and experiment with CORS, we used this on-line tool [22]. This website acts as both, client and server, it allows us to change server settings and request to see the different responses on the different combinations of configuration/request and the response they trigger.

2.5.4 CORS vs XSS and CSRF

At first glance some people might thing that CORS is able to prevent XSS (Cross Site Scripting) or CSRF (Cross Site Request Forgery), but the reality is other, CORS does not prevent XSS or CSRF (at least the major part of this attacks) and it is not meant to do so. The first thing we need to understand is that CORS purpose is NOT to defend us from this attacks or any other, CORS is not a defend mechanism, it might act as such in some cases, but it is not CORS's purpose. CORS purpose is to provide a controlled way to relax the restrictions imposed by the same-origin policy (SOP).

All that said, lets detail more why CORS should not be used to prevent CSRF or XSS. If we look at XSS, the attacker is placing an evil piece of JavaScript that will make a call from a website, which is accepted by CORS, and then exploit that by making request to server.

Then for CSRF, any request generated by an HTML form can be a simple request and will never be preflighted. Form submissions can be sent from any origin to any other origin. The browser will display the response, but in doing so it will change the URL (i.e. origin) to that of the responding domain. There is no way for the sending origin of a cross-origin request to access a form submission response via JavaScript.

While a properly configured CORS policy is important, it does not in itself constitute a defense. Adding a CORS policy where there was none before cannot confer any additional protection against any attack. In fact, an overly permissive CORS policy can undermine some attempts to prevent attacks.

Chapter 3

Project Development

3.1 Token management and verification

The main problem encountered during the realization of this project was the way to generate and store the token used to authenticate the users. First we decided to use JWT since it seems a better option in comparison with session ID. Once we established that we faced various problems when storing the token as explained on chapter 2.4 section 4.4.2 either storing the token on the cookies or on the web storage, local or session, the token could be stolen by an attacker using XSS or CSRF as ilustrated by the following table: 3.1.

	Cookies	Local Storage	Session Storage
Vulnerable to CSRF	Yes	No	No
Vulnerable to XSS	No [1]	Yes	Yes

Table 3.1: Vulnerability comparison of client storage

To face this problems we isolated both attacks, first we approached XSS, since malicious scripts inside your web application will in most cases have an overall impact in it, we tried to at least minimize that impact preventing the access to the user authentication tokens. To do this we used the Secure and HTTPOnly flags explained on chapter 2.1.

As a result, even if a XSS flaw exists, and someone finds and exploits it, the browser will not reveal the cookie to the attacker.

But now once we are using cookies we can be subject to CSRF attack, since the browser will send the cookies on each request, if we access a malicious site it can make a request to a third site with our cookies tricking the server into think we are actually making that request.

We could prevent this using session ID, but can we emulate this kind of behaviour with JWT. To do this we use JWT synchronize pattern and local or session storage. To simulate the session ID we add in the response of the login an other JWT, different from the one set on the cookies, then we store it on the web storage. This way we stop this kind of attack since this JWT will be send on all the request and the server will verify it is the correct one. This process is illustrated on the Figure: 3.1.

[1] This is true if the cookie have the HttpOnly flag active and the browser supports it.

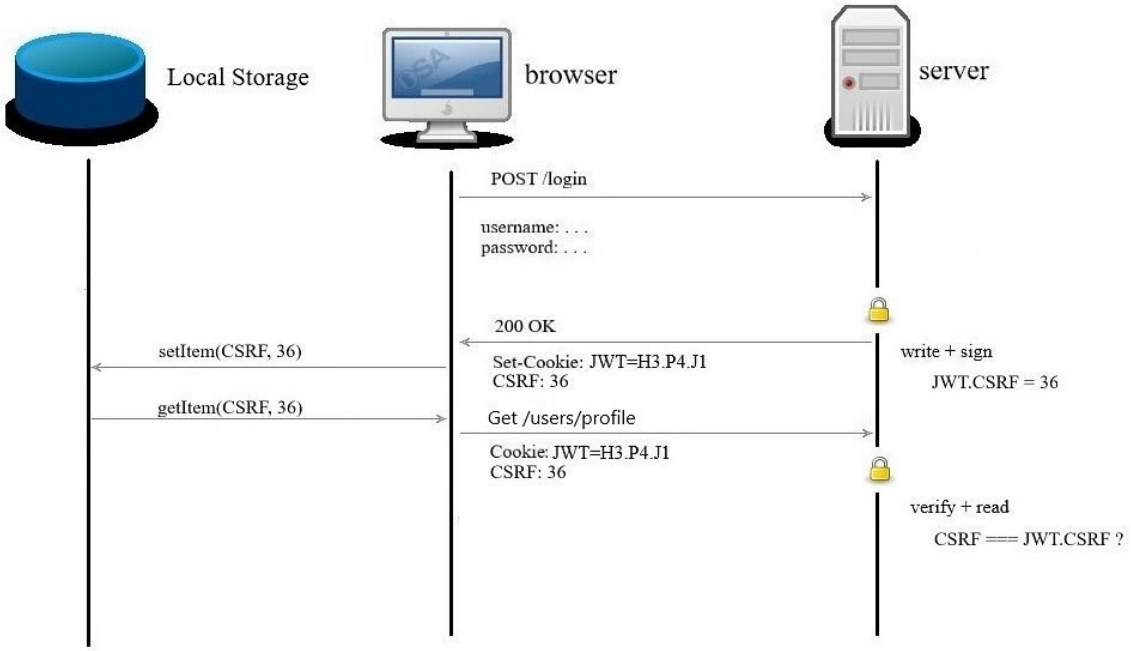


Figure 3.1: Simulating session ID with JWT

Now, we can access this token on the local storage by scripts, but still we will not be able to access the cookies, so we prevented both attacks. We can see the full process on the Figure: 3.2.

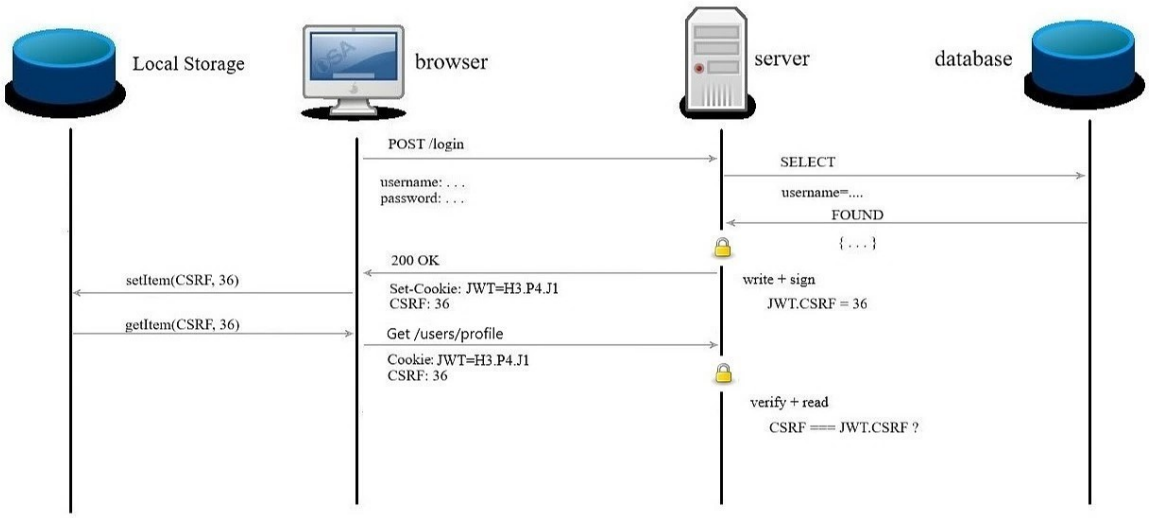


Figure 3.2: Solution implemented on the project

3.2 Back End Implementation

The back end implementation of the project is really important, here is were almost all the logic with the JWTs and the prevention is done, here we will stablish a secure connection to prevent MITM attacks ande we will set up the tokens and they verification to access the routes to prevent XSS and CSRF attacks to impersonate any registered user. The full implementation code can be found on the appendix C.1.

3.2.1 Establishing the connection

The first step was to set a secure connection over HTTPS, since if we do the connection over HTTP no amount of countermeasures to prevent the impersonation and stealing of credentials is relevant since any one could spoof the credentials any time a login or register takes place on the application.

To do that we used openSSL to obtain self-signed certificates. Since our application won't be deployed on the web for the purposes of this project we will not need a real certificate authorized by a certificate authority (CA), but we want the connection to go through HTTPS to do a real development of an application even though the one created in this project is for educational purpose only.

To get our certificates and private keys we used open SSL [23]. The command used to create both the certificate and the private key is the following. Note that this command will not work on a Windows CMD, we will need a Linux terminal or, if we are using Windows, use GIT bash.

```
$ openssl req -x509 -sha256 -nodes -days 365 -newkey rsa:2048 -keyout privateKey.key -out
  certificate.crt
```

This command will provide us with a certificate valid for 365 days. After typing the command some questions about the place and who is emitting this certificate will be asked, it is not important since we will be using our certificates for development purposes and we are the only ones that are going to use them so this not trusted self-signed certificates will be enough.

Once the keys and certificates are on a folder we will open the connection of the server and the database and open a port for the client to do the connection with the server. To see how to code this in NodeJS with mongoose we can see the Figure: 3.3

```
mongoose.connect(db.url, (err, database) => {
  if (err) return console.log(err);
  https.createServer(httpsOptions, app).listen(httpsPort);
});
```

Figure 3.3: Connection with the database and creation of the server.

and to see the variables such as the port we use and the HTTPS secrets (key, certificate, pass phrase) see Figure:3.4

```
const keyFilePath = './config/key.pem';
const certFilePath = './config/cert.pem';

const httpsPort = 8443;

const httpsOptions = {
  key: fs.readFileSync(keyFilePath, 'utf8'),
  cert: fs.readFileSync(certFilePath, 'utf8'),
  passphrase: 'Our not so secret passphrase'
};
```

Figure 3.4: Variables used to set up the HTTPS connection.

As said before, this certificates are validated by ourselves, this is OK for a locally web used as a PoC, if we ever wanted to deploy this page, to prevent MITM attacks we must get the appropriate keys certificated by a proper trusted CA, for a more detailed explanation see the section 2.2.3 on the chapter 2.3.

3.2.2 End points

Once we got a secure connection, we need to set up the end points of our back end to be able to register, login and get a profile to test the correct check of the credentials for the logged user. Before that we will create a mongoose schema for a user that will have four properties. First the name of the "name" field that will be a required string. Then the "username", this field will have the same characteristics as the "name" but it must be unique, there can not be two "username" fields that are the same. After that we will have the field "password" this field will be also a string and required. Finally the "email" field with the exact same characteristics of the "username", string, required and unique. We can see the code for this on the Figure:3.5.

```
let UserSchema = mongoose.Schema({
  name: {
    type: String,
    required: true
  },
  username: {
    type: String,
    required: true,
    unique: true
  },
  password: {
    type: String,
    required: true
  },
  email: {
    type: String,
    required: true,
    unique: true
  }
});
```

Figure 3.5: User schema.

Once we set up the schema we will code our end points, just in case the app is used some time for other purpose we will set our end points on the URL '/users', and inside that we will do '/register', '/login' and '/profile'. Here we illustrate the tree structure of the end points.

```
localhost:8443 (IP adress of server)
└─/users
   ├─/register (POST)
   ├─/login (POST)
   └─/profile (GET)
```

This way it is easier to add more end points either for future lines of this TFG or if we use this PoC to actually deploy a web application. Once we decided how to organize the end points we need to code them, first of all we need to tell the application that '/users' is available, with express this is quite easy as we can see on the Figure: 3.6.

```
const users = require('./routes/users');

app.use('/users', users);
```

Figure 3.6: Middleware executed for '/users' URL.

This piece of code will be executed when a request is made to the URL '/users', we have the child routes inside the folder './routes/users' and this grants access at them with the form we want to (e.g. /users/register).

3.2.3 Register

The '/register' route will add new users to our database, we can see the code on the Figure:3.7.

```
router.post('/register', (req, res) => {
  const user = req.body;
  let newUser = new User({
    name: user.name,
    email: user.email,
    username: user.username,
    password: user.password
  });

  User.addUser(newUser, (err, user) => {
    if (err) {
      if (err['code'] === 11000) {
        res.status(409).json({
          success: false,
          message: 'Failed to register user, duplicated'
        });
      } else {
        res.status(500).json({
          success: false,
          message: 'Failed to register user',
          error: err['message']
        });
      }
    } else {
      res.json({ success: true, message: 'User registered' });
    }
  });
});
```

Figure 3.7: Function called when a POST request is made on '/users/register'.

As we can see on the image, we will call the function "addUser" and pass to it the user we created using the information of the request and a function, we will explain more about this function after seeing "addUser".

The addUser function is really important since is the storage of the user's information, the more sensible field is the password and it is important to think about how to store it. It is important to note that storing passwords is a hard task, is easy to get it wrong, if the possibility of letting others do the job using "sign with * (google, facebook, etc.)" in most cases it will be better to let them handle it because in most cases they will do a better job than a new developer.

Since we will not do that we will look at the multiple options to store the a password.

- First we could just store the passwords in plain text, this is a really bad idea since if someone gains access to our database it is important not to reveal the passwords of our users since the attacker will be able to impersonate them, making our efforts tu authenticate them futile. Maybe more important a lot of people reuses passwords for multiple things and we have an email stored too so an attacker my access that too if the password is the same.

- Another approach is to encrypt it, this is a more secure way to store passwords but it has some big flaws. The first one is that if an insider or even a hacker gains access to the key used for the encryption, they will gain access to all the passwords. The other problem with this method is that if you have a lot of users on the site the same password will have the same encryption and if a lot of people have the same one it must be a common one (e.g. 1234, password, qwerty, etc.) and it will be easy to deduce.

- A third way to store the password is to hash it, this approach have the same problem as encrypting, the same password will have the same hash and more important, there is lists of common password hashes on Google (Rainbow tables), and even hash "breakers" that already did the computation of the hash of billions of the most common passwords and put them into a database and just copying and pasting a hashed password there, if it is common, it will most likely break it in a few seconds. To prevent this we can salt the passwords (Add a random string of characters to it), this way all passwords will be different and most likely not on a rainbow table preventing the most common attacks.

On our application we use the bcrypt package for NodeJS to hash and salt our passwords using the bcrypt algorithm. We use this since the research done lead to it because it is one of the most secure ways of hashing and salting nowadays.

We can see the implementation of the "addUser" function on the Figure:3.8.

```
module.exports.addUser = function(newUser, callback) {
  bcrypt.genSalt(10, (err, salt) => {
    bcrypt.hash(newUser.password, salt, (err, hash) => {
      if (err) throw err;
      newUser.password = hash;
      newUser.save(callback);
    });
  });
};
```

Figure 3.8: Function used to store users to the database.

We can see that after storing the data to the database we call the function inside '/register' as a callback, see Figure:3.6. This function is important since it will be communicating the client what happened with the register.

There is multiple possibilities:

- addUser returns an error code 11000, this means that one of the parameters flagged as unique it's already inside the data base. If that is the case we will return the client an error 409 (Conflict) and a message that makes the client notice that the user is duplicated.
- addUser returns an error different of 11000. In that case we will return an error 500 (Internal Server Error) and a message telling the client the user failed to be registered.
- addUser adds the user successfully to the database. If this happens we will inform the client the user was successfully registered.

3.2.4 Login

The '/login' route will check if the credentials (username & password) send to the server by the client match any one stored in our database. This route is really important since it will set up the tokens for the correct authentication of the user. We can see the code used to implement the '/login' on the Figure:3.9.

```
router.post('/login', (req, res) => {
  const username = req.body.username;
  const password = req.body.password;
  User.getUserByUsername(username, (err, user) => {
    if (err) throw err;
    if (!user) {
      return res
        .status(404)
        .json({ success: false, message: 'User not found' });
    }
    User.comparePassword(password, user.password, (err, isMatch) => {
      if (err) throw err;
      if (isMatch) {
        const token = jwt.sign(user.toJSON(), config.secret, {
          expiresIn: 21600 // 6h
        });
        const csrfToken = jwt.sign(user.toJSON(), config.secretSession, {
          expiresIn: 21600 // 6h
        });
        res.cookie('jwt', token, {
          expires: new Date(Date.now() + 21600000), // 6h
          secure: true,
          httpOnly: true
        });
        res.json({
          success: true,
          message: 'Logged in',
          csrfToken: csrfToken
        });
      } else {
        return res
          .status(400)
          .json({ success: false, message: 'Wrong password' });
      }
    });
  });
});
```

Figure 3.9: Function called when a POST request is made on '/users/login'.

As we can see, we first call the function "getUserByUsername", this function will take the username of the request and look for a match on the database, if the user is not on the database, the server will return an error 404 (Not found) and a message informing the client that the username provided was not found. If the user is found the function will call "comparePassword" that will compare the password provided on the request and the one stored on the database. If the passwords do not match the server will return to the client an error 400 (Bad Request) and a message informing the password was incorrect. If the passwords match the client will respond the client with a message to inform the user credentials were correct and will also send the cookie with the JWT with the flags "HTTPOnly" and "Secure" to prevent XSS and another token within the response message that the client will store on the web storage to prevent CSRF. Both tokens and the cookie have a 6 hours expiration time, this time should match the token purpose and have a balance between security and usability. For example a one time token used for a banking application should have a short expiration time (5 - 10 minutes) because is heavily focused on the security, since our application is just used for a PoC we set 6 hours, just to demonstrate that is possible to set the expiration time and to show how it is done.

We can see the implementation of the functions "getUserByUsername" and "comparePassword" on the Figure:3.10.

```
module.exports.getUserByUsername = function(username, callback) {
  const query = { username: username };
  User.findOne(query, callback);
};

module.exports.comparePassword = function(candidatePassword, hash, callback) {
  bcrypt.compare(candidatePassword, hash, (err, isMatch) => {
    if (err) throw err;
    callback(null, isMatch);
  });
};
```

Figure 3.10: "getUserByUsername" & "comparePassword" implementation.

As we can see we use the package bycript to compare the password provided (candidatePassword) and the password stored, already hashed (hash).

3.2.5 Profile

The '/profile' route is purely for testing our token set up, it uses the passport package to check the token (CSRF token) send by the request is a correct one, if it is it will response the client with information about the logged user. We can see the implementation of this on the Figure:3.11.

```
router.get(
  '/profile',
  passport.authenticate('jwt', { session: false }),
  (req, res) => {
    res.json({
      name: req.user.name,
      username: req.user.username,
      email: req.user.email
    });
  }
);
```

Figure 3.11: Function called when a GET request is made on '/users/profile'.

3.2.6 Back end interceptor

We saw the check of the CSRF token on the '/users/profile' route, this was an early implementation with only one token, after we realized that was not secure enough we added the new token inside the cookies. After that we also realised that check for this two tokens inside all the required routes would not be the ideal for two big reasons. First from a developer point of view to repeat the same code every time we need to check the tokens would be bad practice, so we would create a separate function that did that and then put it inside every route that needs the token check, but that leads to the second reason. If we have a lot

of routes on our web application, usually the case on bigger web pages, it would be easy to miss one and that could create a big hole on our security.

After notice that we decided to create a function that will be executed for any request send to our API and inside it exclude the routes that do not need authorisation to be accessed, the registration and login routes on our case. We can see the implementation of this function on the Figure:3.12.

```
app.use((req, res, next) => {
  const token = req.headers.authorization;
  if (req.url === '/users/login' || req.url === '/users/register') {
    next();
  } else if (token && req.cookies.jwt) {
    jwt.verify(token, db.secretSession, (err, decoded) => {
      if (err) {
        if (err.name === 'TokenExpiredError') {
          return res.status(403).json({
            success: false,
            message: 'Token expired please log in again',
            logout: true
          });
        } else {
          return res.status(403).json({
            success: false,
            message: 'Failed to authenticate token'
          });
        }
      } else {
        jwt.verify(req.cookies.jwt, db.secret, (err, decoded) => {
          if (err) {
            if (err.name === 'TokenExpiredError') {
              return res.status(403).json({
                success: false,
                message: 'Token expired please log in again',
                logout: true
              });
            } else {
              return res.status(403).json({
                success: false,
                message: 'Failed to authenticate token'
              });
            }
          } else {
            next();
          }
        });
      }
    });
  } else {
    res.status(401).json({ success: false, message: 'Unauthorized access' });
  }
});
```

Figure 3.12: Function called for any request to our API (Back end interceptor).

As we can see, this function will first check if the request URL is for the routes '/users/login' or '/users/register', if that is the case it will not check for the tokens and will let the request go to those routes. If the URL is not any of those two the check for the authorisation credentials will start following the next steps:

- First it will look if there are tokens on the headers and on the cookies, if one of them is missing, the server will respond with an error 401 (Unauthorized) and a message informing the client the access was unauthorized.
- If both tokens are present on the request, then the function will check for the token send with the headers (CSRF token) if it is correct, then will check for the cookies (XSS token) if this is also correct the function will let the request go to the route.
- If there is an error with any one of the tokens this function will check two possibilities, in our case are the same checks in both cases:
  - If the error name is "TokenExpiredError", that means the token send is expired, the function will send to the client a response with the error 403 (Forbidden), a message asking the user to re-log is and a boolean to tell the client to log out said user.
  - If any other error is presented the same error will be send 403 (Forbidden) but the message will be telling the client that the token was wrong, so the server could not authenticate it.

3.3 Front End Implementation

Even if the back end implementation takes almost all of the burden with the security checks of the authentication and authorisation process, the front end will be the responsible to store the tokens correctly and send them when needed, also we need to prevent unnecessary calls to the back end protected routes when we know the result will turn in an error response (e.g. request a profile without sending tokens). The full implementation code can be found onthe appendix C.2.

3.3.1 Interceptor

Same as the back end, to prevent repetition of code and reduce the human error possibility to a minimum, we made a service, an interceptor service to be precise, that will take all the request made to our server, check if there is a token with the key "idToken" stored on the session storage and if there is it will clone the request, add a header "Authorization" with the value of "idToken" and send the request to the service, this way we will always have the token on our header, if there is no token it will send the request without the "Authorization" header. We can see the implementation on the Figure: 3.13.

```
import { Injectable } from '@angular/core';
import {
  HttpInterceptor,
  HttpRequest,
  HttpHandler,
  HttpSentEvent,
  HttpHeaderResponse,
  HttpProgressEvent,
  HttpResponse,
  HttpUserEvent
} from '@angular/common/http';
import { Observable } from 'rxjs/Observable';

@Injectable()
export class InterceptorService implements HttpInterceptor {
  intercept(
    req: HttpRequest<any>,
    next: HttpHandler
  ): Observable<
    | HttpSentEvent
    | HttpHeaderResponse
    | HttpProgressEvent
    | HttpResponse<any>
    | HttpUserEvent<any>
  > {
    const idToken = sessionStorage.getItem('idToken');
    if (idToken) {
      const cloned = req.clone({
        headers: req.headers.set('Authorization', idToken)
      });
      return next.handle(cloned);
    } else {
      return next.handle(req);
    }
  }
  constructor() {}
}
```

Figure 3.13: Front end interceptor.

3.3.2 Auth

The Auth service is the responsible to manage the calls to the service, in our case to make the post calls for register and login and to make the get call on the profile. This service will also check if there is a token in the session storage. We can see the

implementation of this service on the Figure:3.14.

```
import { Injectable } from '@angular/core';
import { HttpClient, HttpHeaders } from '@angular/common/http';
import { Observable } from 'rxjs/Observable';

// model
import { UserResponse } from '../../core/models/userResponse.model';
import { User } from '../models/user.model';

@Injectable()
export class AuthService {
  baseURL = 'https://localhost:8443';
  constructor(private http: HttpClient) {}

  registerUser(user) {
    const headers: HttpHeaders = new HttpHeaders();
    headers.append('Content-Type', 'application/json');
    return this.http
      .post(this.baseURL + '/users/register', user, { headers: headers })
      .map(res => res);
  }

  authenticateUser(user) {
    const headers = new HttpHeaders();
    headers.append('Content-Type', 'application/json');
    return this.http
      .post(this.baseURL + '/users/login', user, {
        headers: headers,
        withCredentials: true
      })
      .map(res => {
        return res;
      });
  }
```

```
  getProfile() {
    const headers = new HttpHeaders({
      'Content-Type': 'application/json'
    });
    return this.http
      .get(this.baseURL + '/users/profile', {
        headers: headers,
        withCredentials: true
      })
      .map((res: User) => {
        return res;
      });
  }

  loggedIn() {
    if (sessionStorage.getItem('idToken')) {
      return true;
    }
    return false;
  }
}
```

Figure 3.14: Auth service.

It is important to mention that angular request will not send or store cookies if we do not add the option "withCredentials: true".

3.3.3 Validate

The validate service contains a function that will make sure the email the user types on the register matches the pattern XXXX@XXX.X, we can se the function with the regular expression used to accomplish that on the Figure:3.15.

```
@Injectable()
export class ValidateService {
  constructor() {}

  validateEmail(email: String) {
    const re = /^(([^<>()\[\]\\.,;:\s@"]+(\.[^<>()\[\]\\.,;:\s@"]+)*)|(".+"))@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\])|(([a-zA-Z\-0-9]+\.)+[a-zA-Z]{2,}))$/;
    return re.test(String(email).toLowerCase());
  }
}
```

Figure 3.15: Validation service.

3.3.4 Components

On this section we will focus on the components that take part on the security aspect of the application either are part of the work flow of the credentials and tokens (login, register, profile) or prevent access depending if the user is logged in (navigation bar, guards).

3.3.5 Register

3.3.6 Login

The login component is a simple form with two fields, user name and password and a button. When the user pushes the button a POST request is triggered to the end point '/users/login' using the Auth service function "authenticateUser". To prevent any of the fields of the form to be empty and trigger a request with empty fields to the server we placed validators so if any of the fields is empty it will not call the service but it will tell the user to fill all the fields of the form. If any error occurs, a message will get displayed to let the user know what happend (Wrong user name, password, empty field, unexpected error ...).

3.3.7 Profile

The profile component is the component used for testing the tokens, if we go to the route of this component it will make a GET request to the user using the "getProfile" function, with this request both tokens will be send, if there is an error the page will not display any information from the user, but if the authentication process is correct, the name, user name and email will be displayed.

3.3.8 Navbar

The navbar component contains the navigation bar of the application and it have two states. The first one is when the user is not logged in, it will let the user navigate to three places, home, login and register as we can see on the Figure:3.16.

Figure 3.16: Navigation bar if user is not logged in.

The second state is when the user is already logged in the application and it will allow him to navigate to multiple places, almost all the application except the login and the register, and it will have a button to log out the application as we can see on the Figure:3.17

Figure 3.17: Navigation bar if user is logged in.

3.3.9 Guards

Finally the guards is as the name indicates, a guardian for specific routes, in our case if some user without authorisation to access, for example, the route /profile and instead of registering and logging in he types on the URL bar "https://*/profile" (Or any other protected route), the page will redirect him to the home of the application preventing possible buggy calls as the profile route needs user data to do the GET request and at the same time preventing unnecessary requests even if they are not a cause of malfunction.

Chapter 4

Results

The security measures and protocols proposed to secure an SPA against user impersonation and credentials stealing using XSS and CSRF are working.

First, the storage is not accessible from another page, preventing CSRF, and since our cookie is set with HttpOnly we cannot access it from javascript, preventing XSS as we can see on the Figure:4.1.

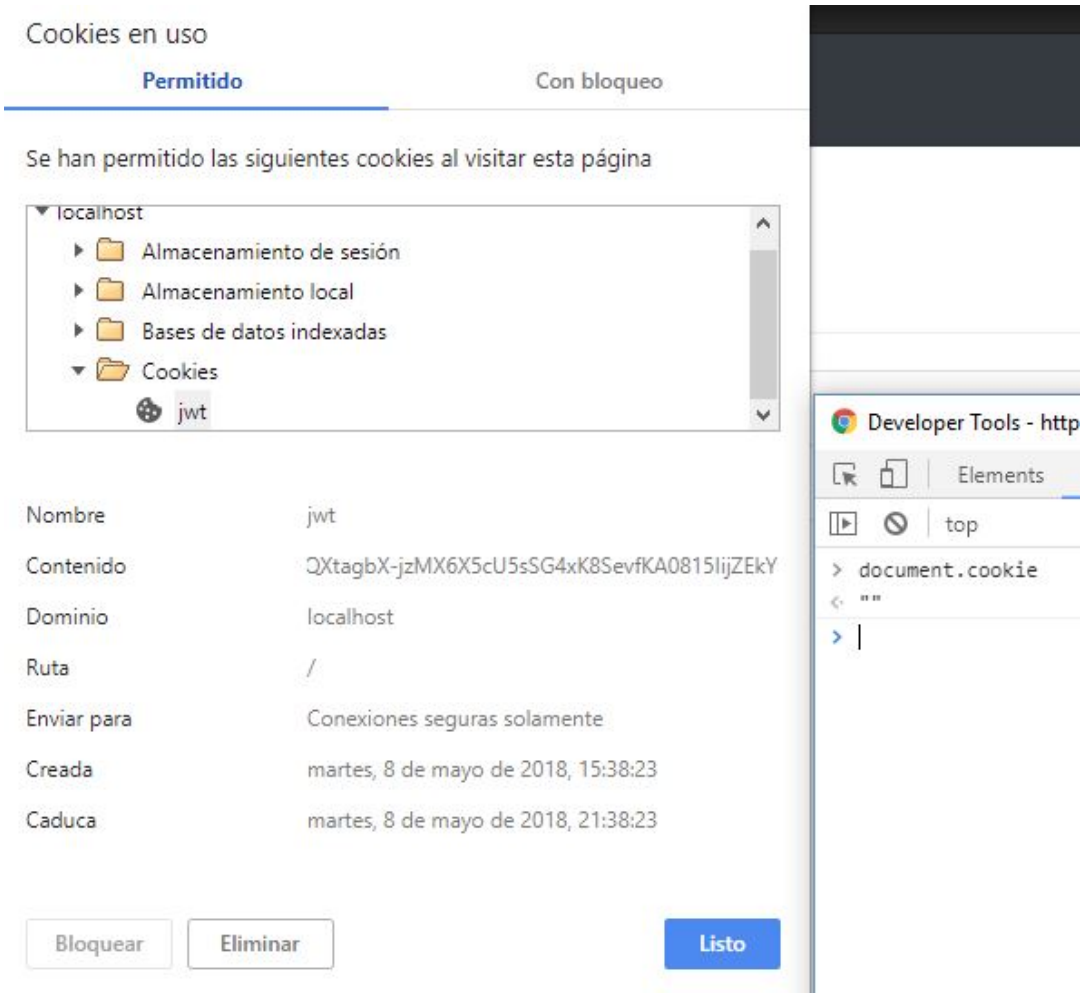


Figure 4.1: Cookie not displayed on document.cookies.

Then, the service will not serve protected information unless both authentication tokens are on the request as we can see on the Figures: 4.2 and 4.3.

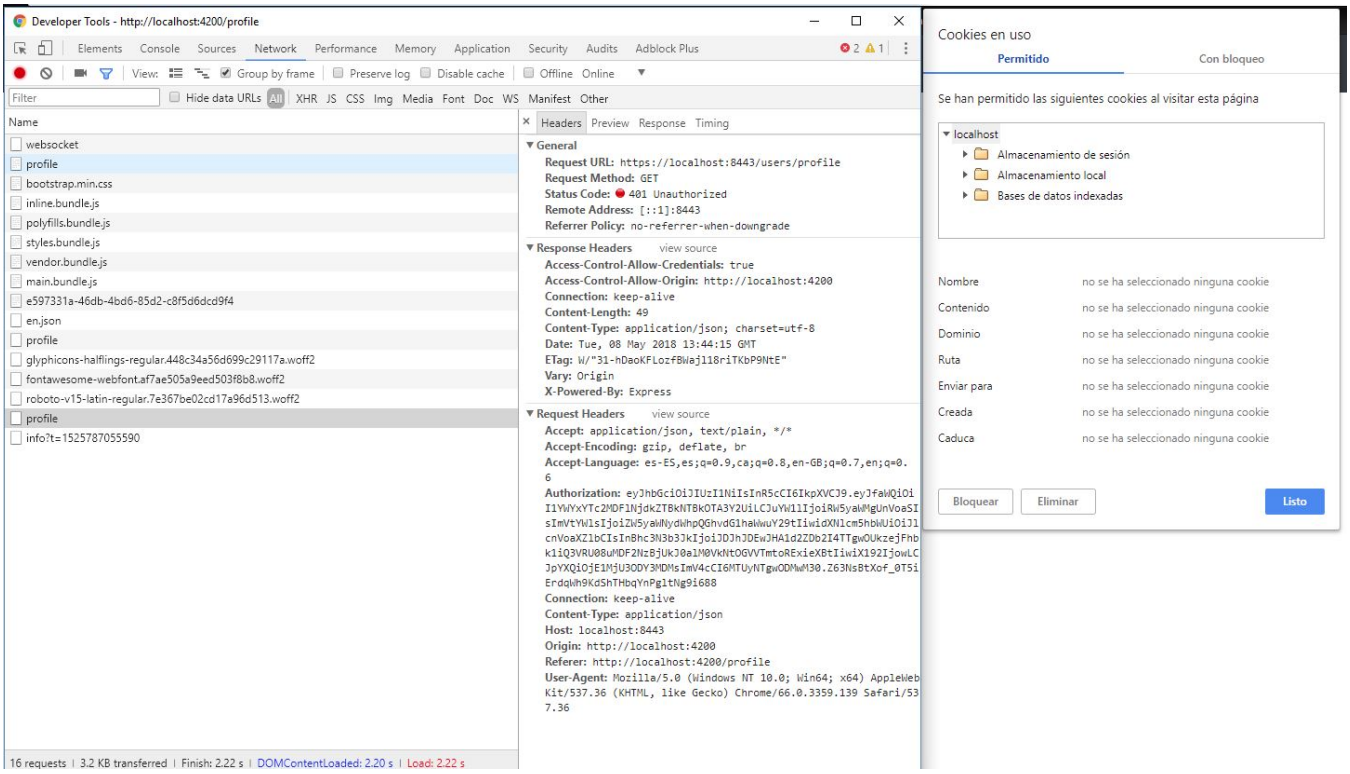


Figure 4.2: Request to /profile with only the Authorisation token (no cookie).

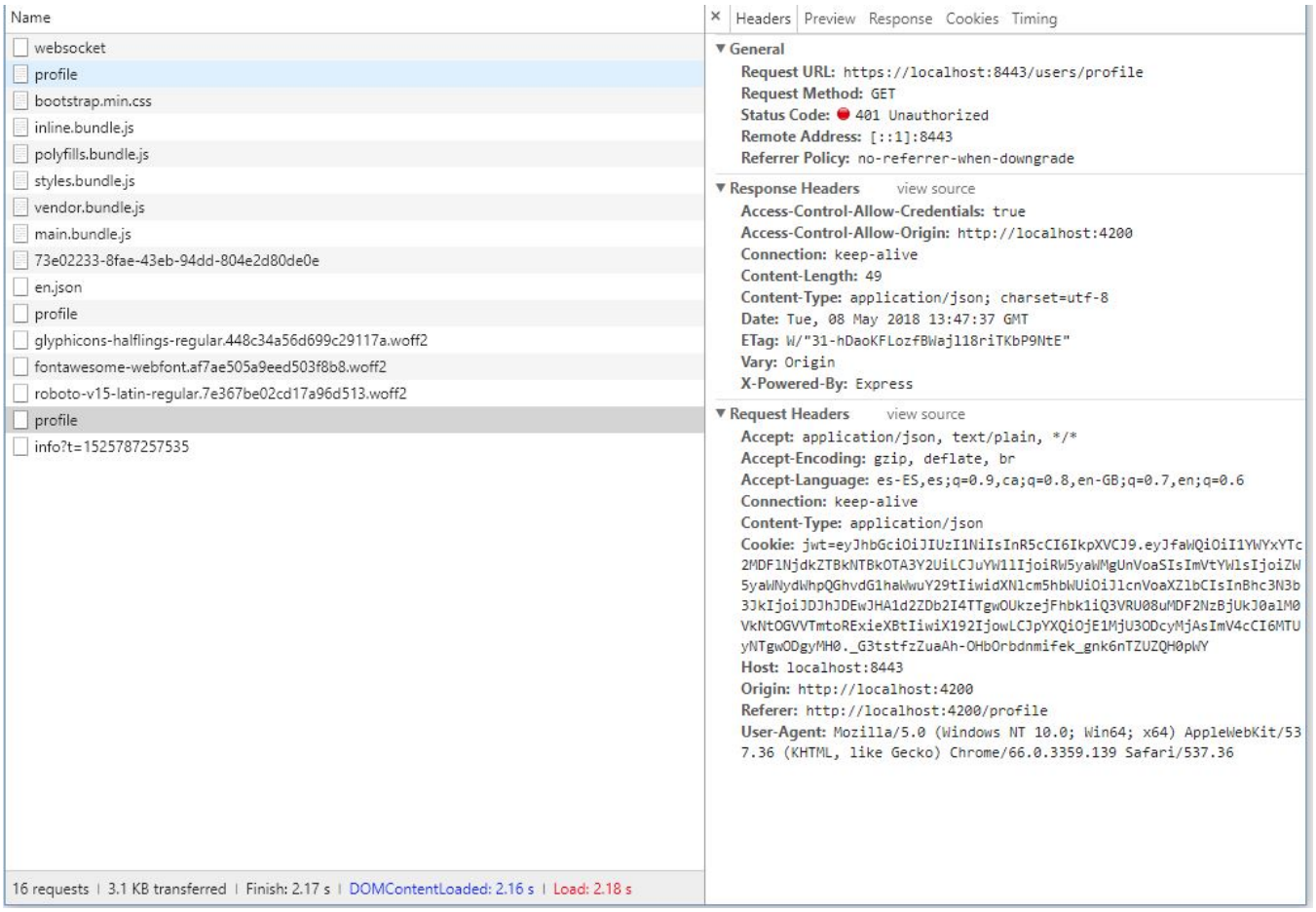


Figure 4.3: Request to /profile with only the cookie (no token).

The service responds with a 401 Unauthorized on both cases as expected.

On the other hand, the service will respond with the user's name, username and email if both tokens are sent correctly, as we can see on the Figure: 4.4.

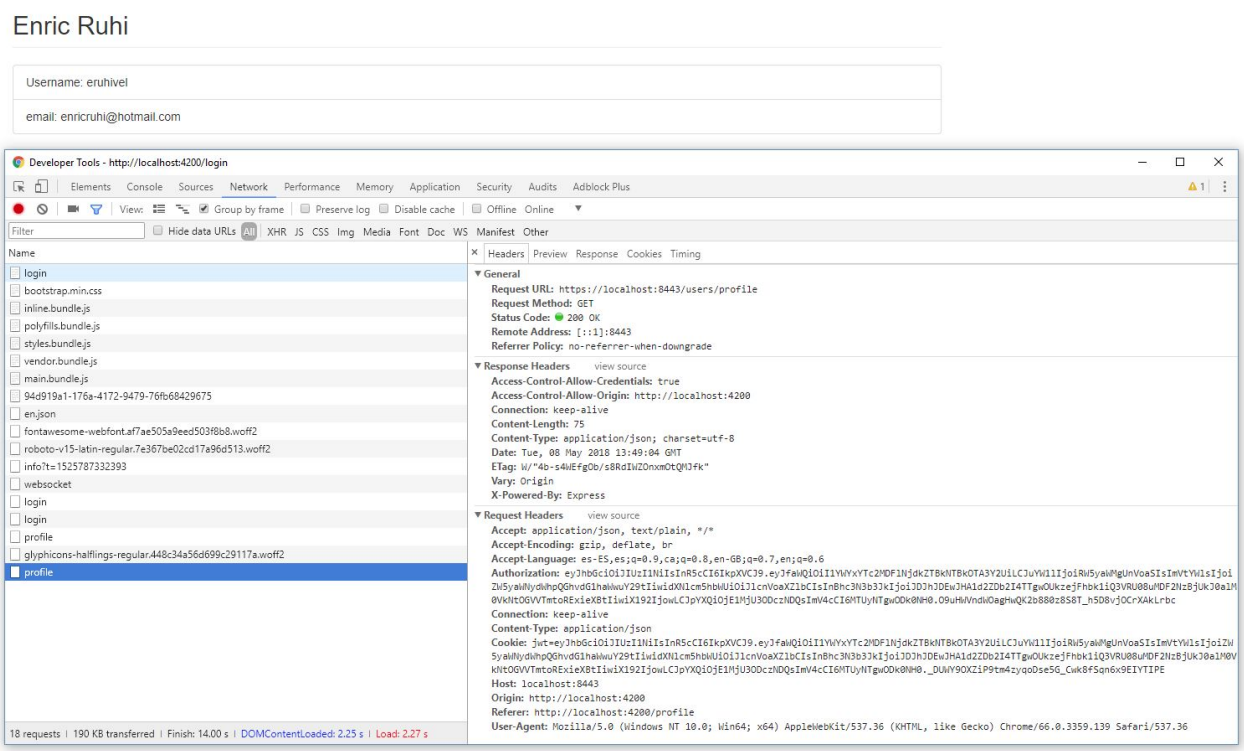


Figure 4.4: Request to /profile with both tokens.

Also, we can see on the Figure:4.5 that the password used, in our test case 123456 is displayed as a salted hash of itself so, if someone gains access to our database, first they will not be able to see the password in plain text, and then it will not break with a rainbow table attack since it will not match a hashed version of itself because of the salt, as we can see on the Figure: 4.6, that shows 123456 hashed using bcrypt without salting it first.

Figure 4.5: Hashed and salted password on the database.

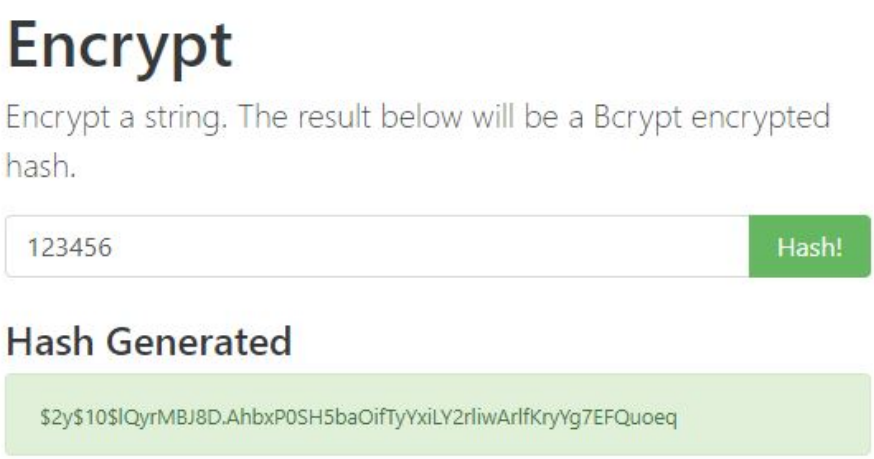


Figure 4.6: Hashed password using Bcrypt without salt.

Chapter 5

Budget

This project was developed using open source tools such as visual studio code, Bibucket (Free edition), Latex, etc.

We will consider the salary of a junior engineer that is 21.000€ considering a 40 hour/week and 4 weeks/month with 14 pays, the salary per hour is 9,37€/h. The duration of the project was 18 weeks with a mean of 35 hours for week.

Concept	**Cost**
Junior engineer	5903,10 €

Chapter 6

Conclusions and future development

Conclusion

This project allowed us to confirm that impersonation and stealing of credentials (e.g. password, email, username) of users is a serious threat. It affects many users every year and it is hard to prevent. This threat affects the full stack of a web application, from the database to the browser and it is easy for a developer to overlook problems or think that a problem is solved when it is not, this is usually because misinformation or lack of knowledge.

After the research done on the thesis we can make various statements:

- All the communication should be transmitted with a secure protocol (HTTPS); if not any one looking at the it, will be able to see all the communication easily, including the user credentials.
- A password, should be salted and hashed before storing it on the database, even so if the password is easy enough (e.g. 123456) it can still be easily cracked.
- The use of authentication tokens (session ID, JWT) does, for itself, not prevent XSS and CSRF, a proper workflow and store mechanisms, must be used to prevent them.
- All the inputs should be escaped to prevent XSS on the page.

All that said, there is always errors or an overlooking; even big companies make mistakes the most recent example is twitter [24]. Furthermore, some attacks are almost impossible to prevent, one example is DigiNotar, a certificate authority that issued rogue certificates for a number of domains, including Google [25].

Finally, during the realization of this project we found a XSS exploit, we can see it on the Figure: 6.1. We should mention that this vulnerability has been reported to the corresponding site administrators.

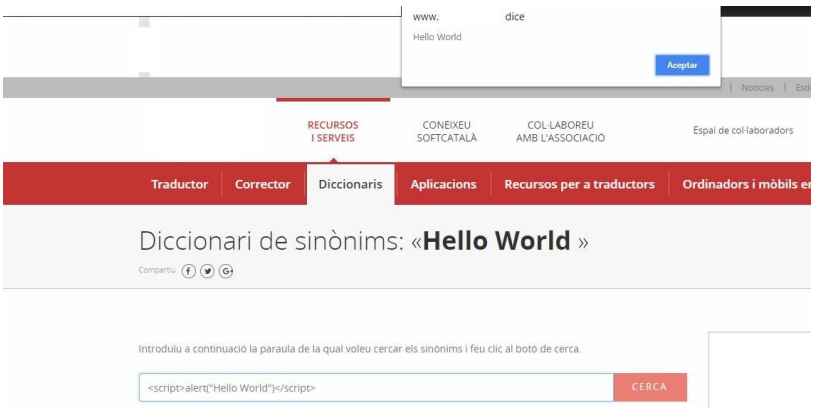


Figure 6.1: Characters not escaped allows <script> to execute.

Future development

There are multiple options for future development of this thesis.

The first is to further develop the application and further research the MEAN stack and do a fully develop web page for some specific purpose.

Another option is to keep researching more about security; the scope of this project is just the tip of the iceberg on web security world, so there are many possibilities. Next, we list some of these possibilities:

- Test further the security of the PoC done on the project and patch the possible issues using variations of the attacks and specific hacking software such as Kali Linux.
- Keep developing the application security, using for example Oauth2.
- Further develop the login security adding a two-factor authentication, also referred to as two-step verification.

In addition, it is possible to make the application to teach basic security to developers and use purposely made mistakes to illustrate various attacks, how they affect the application and/or the users and how to avoid these mistakes.

Bibliography

[1] Http cookie, accessed on 19 February 2018. https://en.wikipedia.org/wiki/HTTP_cookie.

[2] Set-cookie, accessed on 19 February 2018. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie.

[3] Httponly - owasp, accessed on 29 February 2017. https://www.owasp.org/index.php/HttpOnly#Web_Application_Firewalls.

[4] A. Barth. HTTP State Management Mechanism. RFC 6265 (Proposed Standard), April 2011.

[5] Web storage, accessed on 18 February 2018. https://en.wikipedia.org/wiki/Web_storage.

[6] Client-side storage, accessed on 18 February 2018. https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Client-side_web_APIs/Client-side_storage.

[7] Indexeddb api, accessed on 20 February 2018. https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API.

[8] Cross-site request forgery (csrf) - owasp, accessed on 24 January 2018. https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF).

[9] Cross-site scripting (xss) - owasp, accessed on 23 January 2018. https://www.owasp.org/index.php/Cross-site_Scripting_(XSS).

[10] Types of cross-site scripting - owasp, accessed on 23 January 2018. https://www.owasp.org/index.php/Types_of_Cross-Site_Scripting.

[11] Dom based xss - owasp, accessed on 23 January 2018. https://www.owasp.org/index.php/DOM_Based_XSS.

[12] How to test reflected cross site scripting vulnerability, accessed on 07 May 2018. https://www.hackingloops.com/how-to-test-reflected-cross-site-scripting-vulnerability/.

[13] Angular docs security, accessed on 23 January 2018. https://angular.io/guide/security.

[14] Man-in-the-middle attack - owasp, accessed on 24 January 2018. https://www.owasp.org/index.php/Man-in-the-middle_attack.

[15] Http, accessed on 6 November 2017. https://developer.mozilla.org/en-US/docs/Web/HTTP.

[16] Http to https | what is a https certificate, accessed on 7 November 2017. https://www.instantssl.com/ssl-certificate-products/https.html.

[17] Application programming interface, accessed on 3 November 2017. https://en.wikipedia.org/wiki/Application_programming_interface.

[18] M. Jones, J. Bradley, and N. Sakimura. JSON Web Token (JWT). RFC 7519 (Proposed Standard), May 2015. Updated by RFC 7797.

[19] Jwt.io, accessed on 17 February 2018. https://jwt.io/#debugger.

[20] Jwt.io - json web tokens introduction, accessed on 17 February 2018. https://jwt.io/introduction/.

[21] Same-origin policy, accessed on 16 March 2018. https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy.

[22] Test cors, accessed on 16 March 2018. https://www.test-cors.org/.

[23] Openssl, accessed on 4 January 2018. https://www.openssl.org/.

[24] Twitter support on twitter, accessed on 03 May 2018. https://twitter.com/TwitterSupport/status/992132808192634881.

[25] Hackers stole google ssl certificate, dutch firm admits, accessed on 06 May 2018. https://www.computerworld.com/article/2510797/security0/hackers-stole-google-ssl-certificate--dutch-firm-admits.html.

[26] What is mongodb?, accessed on 29 Octover 2017. https://www.mongodb.com/what-is-mongodb.

[27] Mongodb, accessed on 29 Octover 2017. https://docs.mongodb.com/getting-started/shell/.

[28] Install mongodb community edition — mongodb manual 3.6, accessed on 29 Octover 2017. https://docs.mongodb.com/manual/administration/install-community/.

[29] Databases and collections — mongodb manual 3.6, accessed on 29 Octover 2017. https://docs.mongodb.com/manual/core/databases-and-collections/.

[30] Documents — mongodb manual 3.6, accessed on 29 Octover 2017. https://www.mongodb.com/what-is-mongodb.

[31] Bson types — mongodb manual 3.6, accessed 29 on Octover 2017. https://docs.mongodb.com/manual/reference/bson-types/.

[32] Getting started with mongodb (mongodb shell edition) — getting started with mongodb 3.6.0, accessed 29 on Octover 2017. https://docs.mongodb.com/getting-started/shell/.

[33] Angular docs, accessed on 16 March 2018. https://angular.io/guide/docs.

[34] Angular docs, accessed on 7 March 2018. https://angular.io/guide/quickstart.

[35] Single-page application, accessed on 23 March 2018. https://en.wikipedia.org/wiki/Single-page_application.

[36] Node.js Foundation. Node.js, [accessed on 25 January 2018]. https://nodejs.org/en/.

[37] Node.js Foundation, accessed on 25 January 2018. https://nodejs.org/en/about.

[38] Installing node.js via package manager | node.js, accessed on 2 February 2018. https://nodejs.org/en/download/package-manager/#debian-and-ubuntu-based-linux-distributions.

[39] kelektiv/node.bcrypt.js, accessed on 05 Frebuary 2018. https://github.com/kelektiv/node.bcrypt.js.

[40] Express - node.js web application framework, accessed on 29 January 2018. http://expressjs.com/.

[41] Writing middleware for use in express apps, accessed on 29 January 2018. http://expressjs.com/en/guide/writing-middleware.html/.

[42] expressjs/body-parser, accessed on 02 Frebuary 2018. https://github.com/expressjs/body-parser/.

[43] expressjs/cookie-parser, accessed on 02 Frebuary 2018. https://github.com/expressjs/cookie-parser/.

[44] expressjs/cors, accessed on 02 Frebuary 2018. https://github.com/expressjs/cors/.

[45] auth0/node-jsonwebtoken, accessed on 12 Frebuary 2018. https://github.com/auth0/node-jsonwebtokens.

[46] Mongodb node.js driver. http://mongodb.github.io/node-mongodb-native/3.0/.

[47] Introduction to mongoose for mongodb, accessed on 3 Frebuary 2018. https://medium.freecodecamp.org/introduction-to-mongoose-for-mongodb-d2a7aa593c57.

[48] Mongoose odm v5.0.4, accessed on 6 Frebuary 2018. http://mongoosejs.com.

[49] Passport.js, accessed on 5 March 2018. http://www.passportjs.org/docs/.

[50] themikenicholson/passport-jwt, accessed on 6 March 2018. https://github.com/themikenicholson/passport-jwt.

[51] Primeng, accessed on 16 March 2018. https://www.primefaces.org/primeng/#/.

[52] Font awesome, accessed on 14 March 2018. https://en.wikipedia.org/wiki/Font_Awesome.

[53] Bootstrap (front-end framework), accessed on 15 March 2018. https://en.wikipedia.org/wiki/Bootstrap_(front-end_framework).

[54] Bootstrap examples, accessed on 16 March 2018. https://getbootstrap.com/docs/4.0/examples/.

[55] Bootstrap utilites, accessed on 15 March 2018. https://getbootstrap.com/docs/4.0/utilities/flex/.

[56] Vadimdez/ng2-pdf-viewer, accessed on 10 April 2018. https://github.com/VadimDez/ng2-pdf-viewer.

[57] nodemon, accessed on 10 January 2018. https://nodemon.io/.

[58] remy/nodemon, accessed on 10 January 2018. https://github.com/remy/nodemon#nodemon.

[59] /nodemon, accessed on 10 March 2018. https://cli.angular.io/.

Appendix A

MEAN Stack

A.1 MongoDB

A.1.1 Introduction to MongoDB

MongoDB is an open source database that uses document-oriented data model. MongoDB is one of many databases under the NoSQL banner, instead of tables and rows as in relational databases, MongoDB is built on an architecture of collections and documents. Documents comprise sets of key-value pairs and are the basic unit of data in MongoDB. Collections contain sets of documents and function as the equivalent of relational database tables. We can see a comparison of a relational database (e.g. SQL) and the MongoDB data structure on the Figure: A.1.

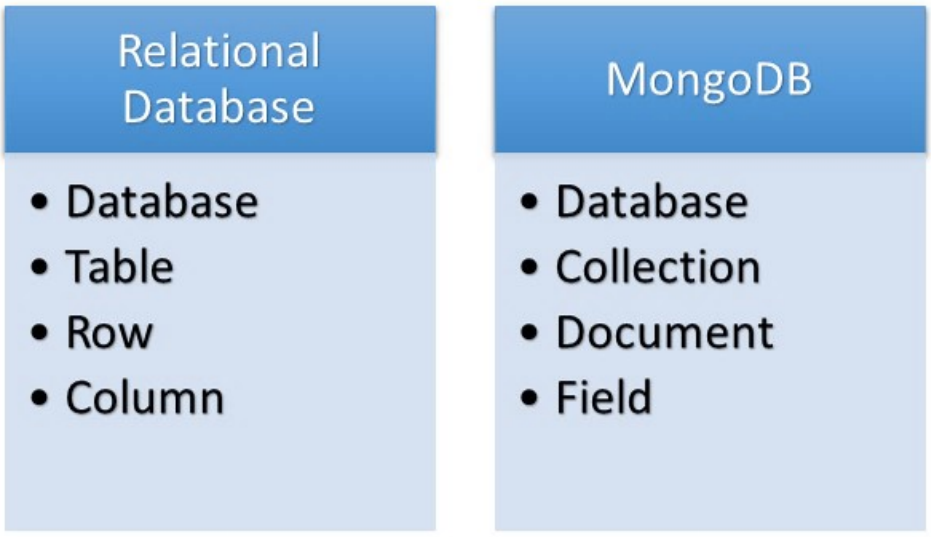


Figure A.1: Data structure comparison.

Like other NoSQL databases, MongoDB supports dynamic schema design, allowing the documents in a collection to have different fields and structures. The database uses a document storage and data interchange format called BSON, which provides a binary representation of JSON-like documents. Automatic sharing enables data in a collection to be distributed across multiple systems for scaling out as data volumes increase. It is important to understand that multiple documents can be inside the same collection and multiple fields can be inside a document as ilustrated by Figure:A.2.

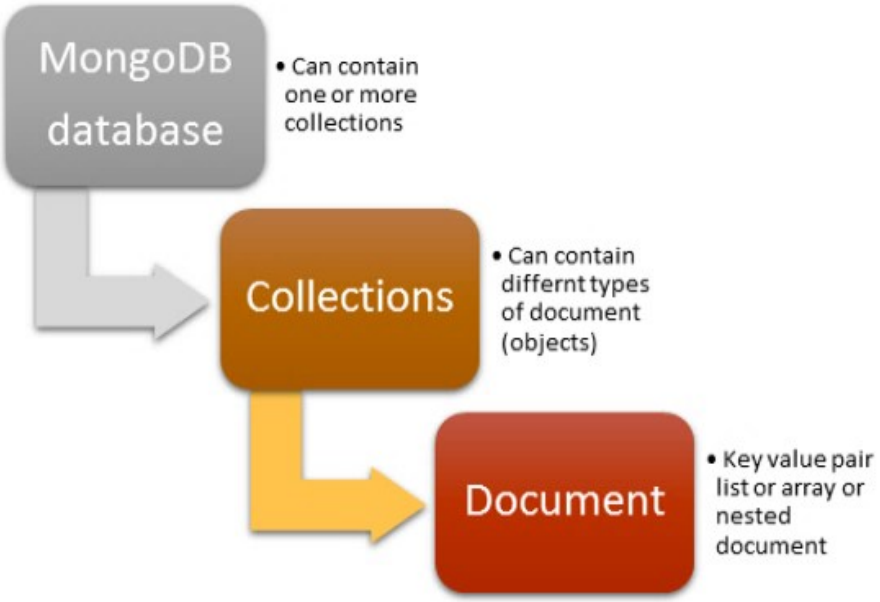


Figure A.2: Data structure of MongoDB.

For more information about MongoDB you can go to their web page [26].

A.1.2 Installation of MongoDB

There are two "distributions" of MongoDB, the community edition (Free) and Enterprise which is available for subscribers and includes additional features. you will be using MongoDB community edition.

Ubuntu 16.04

First of all you will import the public key used by the package management system. Issue the following command to import the MongoDB public GPG key:

```
# sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv
0C49F3730359A14518585931BC711F9BA15703C6
```

Then you will create a list file for MongoDB.

```
# echo "deb [ arch=amd64 ] http://repo.mongodb.org/apt/ubuntu
precise/mongodb-org/3.4 multiverse" | sudo tee
/etc/apt/sources.list.d/mongodb-org-3.4.list
```

After that, reload local package database and install the MongoDB packages.

```
# sudo apt-get update
# sudo apt-get install -y mongodb-org
```

Then you can start MongoDB.

```
# sudo service mongod start
```

Now your MongoDB database is up and running. If you want to verify if MongoDB has started successfully you can do it by checking the contents of the log file.

```
$ cat /var/log/mongodb/mongod.log
```

Look for a line reading "[initandlisten] waiting for connections on port <port>" where <port> is the port configured in /etc/mongod.conf, 27017 by default. You can find the line easier using pipes.

```
$ cat /var/log/mongodb/mongod.log | grep "waiting for connections on port"
```

Finally you can stop the mongod process with the following command:

```
# sudo service mongod stop
```

Windows 10

The windows installation is pretty straight forward. First of all we go to MongoDB main page [27], then we click on downloads on the top right corner, see Figure: A.3.

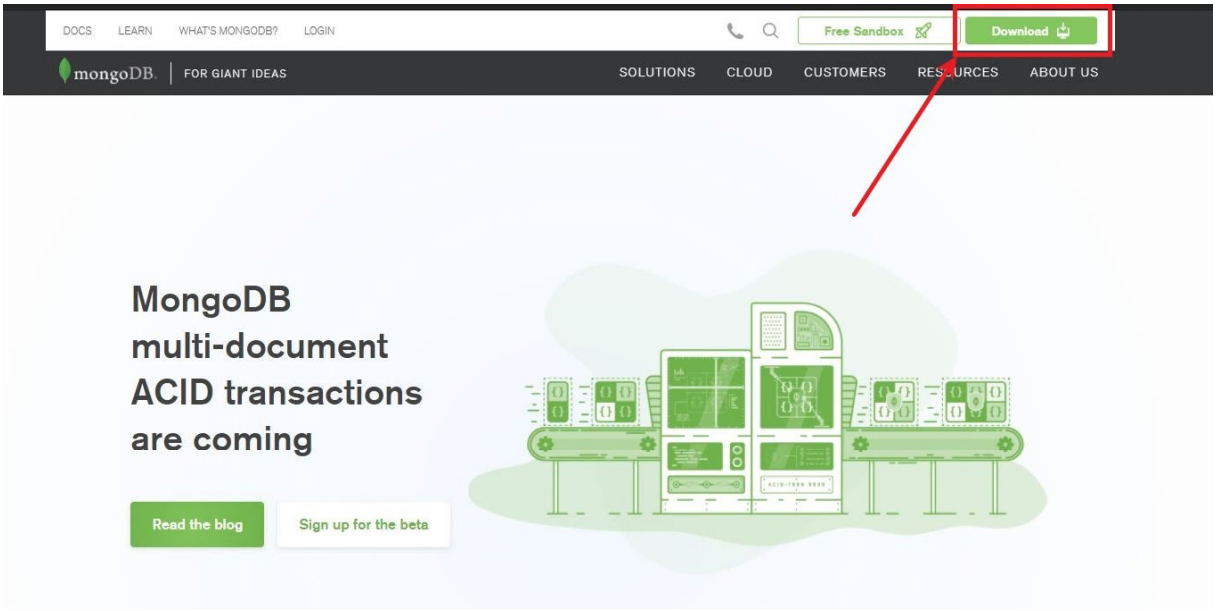


Figure A.3: Download button.

Then we select the "Community Server" version and make sure our operative system is the one listed under the label "Version:", see Figure: A.3

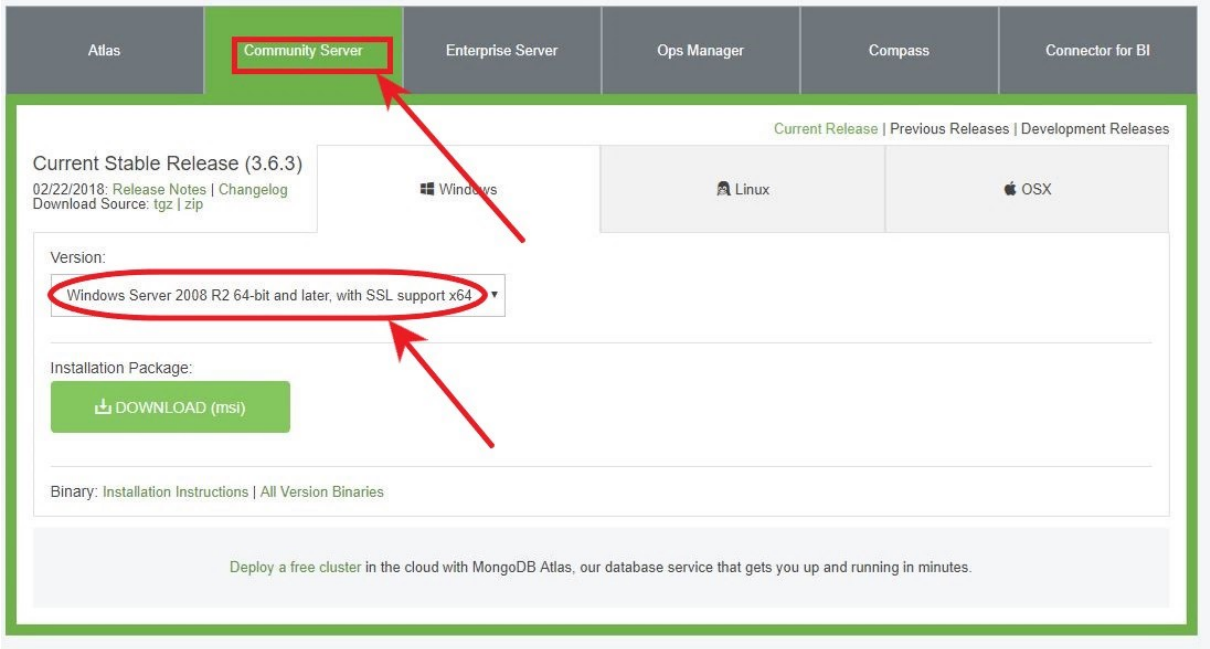


Figure A.4: MongoDB Edition & Operative System selection.

After that a file will be download, we execute it and follow the installation instructions. It is recommended to install the complete version, but if you just want to change to change the path that option is inside the custom option. At the penultimate step there is a check box to install MongoDB Compass, we recommend to install it if you are not comfortable working with terminal since it is a visual user interface that will allow you to see the changes and modify the database from outside the command line.

Other SO

MongoDB is available in many SO, and many distributions. More information about how to install MongoDB on other SO can be found here [28].

A.1.3 Working with mongo shell

To access the MongoDB shell, you will type the following command after starting the mongod process.

```
$ mongo
```

If it does not work, execute the following commands in order:

```
# sudo systemctl enable mongod
# sudo service mongod restart
# sudo service mongod start
$ mongo
```

Once you are inside the shell there are a lot of commands and possibilities, you are going to see some of them, for more information about it you can go to the following MongoDB documentation [29] [30].

Databases

To create a database, you will be using the command **use**. The command will create a new database if it does not exist, otherwise it will return the existing database.

```
> use mydb
switched to db mydb
```

To check your currently selected database, use the command **db**, and if you want to check your databases list, use the command **show dbs**.

```
> db
mydb

> show dbs
admin   0.078GB
local   0.078GB
test    0.078GB
```

Note that your created database(mydb) is not present. To display it you need to insert at least one document into it, you will learn how later.

If you want to delete a database, then **dropDatabase()** command will do the trick.

```
> use mydb
switched to db mydb

> db.dropDatabase()
{ "ok" : 1 }
```

Collections

To create a collection in MongoDB **db.createCollection(name, options)** is the command used to do it. The parameter name is the name of collection to be created. Options is a document and is used to specify configuration of collection (options about memory size and indexing). Options parameter is optional, so you only need to specify the name of the collection.

```
> use mydb
switched to db mydb

> db.createCollection("myCollection")
{ "ok" : 1 }
```

You can check the created collection by using the command show collections.

```
> show collections
myCollection
system.indexes
```

To delete collections you can use the **drop()** command.

```
> use mydb
switched to db mydb

> show collections
myCollection
system.indexes

> db.myCollection.drop()
true

> show collections
system.indexes
```

Documents

To insert data into a MongoDB collection you need to use **insert()** or **save()**. If you attempt to add documents to a collection that does not exist, MongoDB will create it for you. **save()** behaves like **insert()**, unless the *_id* parameter is specified on the document and it exists already on the database, if this conditions are matched, it will update the already existing document.

```
> db.myCollection.insert({title: 'MongoDB document',
description: 'Test document', documentNumber: 1})
WriteResult({ "nInserted" : 1 })
```

TO query data from a collection you need to use **find()** method. **find()** will display the results in a non-structured way, to show them in a formatted way, you can use **pretty()** .

```
> db.myCollection.find()
{ "_id" : ObjectId("5a22e39d80e43e57757a946b"), "title" : "MongoDB document",
"description" : "Test document", "documentNumber" : 1 }

> db.myCollection.find().pretty()
{
        "_id" : ObjectId("5a22e39d80e43e57757a946b"),
        "title" : "MongoDB document",
        "description" : "Test document",
        "documentNumber" : 1
}
```

As you can observe, there is a field that you did not introduce in our document. *_id*. The *_id* field is the unique naming convention that MongoDB uses across all of its content. MongoDB assigns *_id* automatically, you can override it, but is not recommendable unless you know what you are doing.

Documents have the following restrictions on field names:

- The field name *_id* is reserved for use as a primary key; its value must be unique in the collection, is immutable, and may be of any type other than an array.
- The field names cannot start with the dollar sign ($) character.
- The field names cannot contain the dot (.) character.
- The field names cannot contain the null character.

For field value, you can check the BSON types on the MongoDB page [31].

MongoDB's **update()** and **save()** methods are used to update document into a collection. The **update()** method updates the values in the existing document while the **save()** method replaces the existing document with the document passed in **save()** method.

```
> db.myCollection.find().pretty()
{
        "_id" : ObjectId("5a22e39d80e43e57757a946b"),
        "title" : "MongoDB document",
        "description" : "Test document",
        "documentNumber" : 1
}
{
        "_id" : ObjectId("5a22eda180e43e57757a946c"),
        "title" : "document",
        "description" : "Test 2 here",
        "documentNumber" : 2
}
{
        "_id" : ObjectId("5a22edc880e43e57757a946d"),
        "title" : "Last Document",
        "description" : "TODO: Edit this description",
        "documentNumber" : 3
}
```

```
> db.myCollection.update({'title':'Last Document'},{$set:{'description':'New description'}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })

> db.myCollection.find().pretty()
{
        "_id" : ObjectId("5a22e39d80e43e57757a946b"),
        "title" : "MongoDB document",
        "description" : "Test document",
        "documentNumber" : 1
}
{
        "_id" : ObjectId("5a22eda180e43e57757a946c"),
        "title" : "document",
        "description" : "Test 2 here",
        "documentNumber" : 2
}
{
        "_id" : ObjectId("5a22edc880e43e57757a946d"),
        "title" : "Last Document",
        "description" : "New description",
        "documentNumber" : 3
}
```

Finally to delete a document you have **remove()**.

```
> db.myCollection.remove({'documentNumber':1})
WriteResult({ "nRemoved" : 1 })

> db.myCollection.find().pretty()
{
        "_id" : ObjectId("5a22eda180e43e57757a946c"),
        "title" : "document",
        "description" : "Test 2 here",
        "documentNumber" : 2
}
{
        "_id" : ObjectId("5a22edc880e43e57757a946d"),
        "title" : "Last Document",
        "description" : "New description",
        "documentNumber" : 3
}
```

For more information about MongoDB shell, you can check their documentation [32].

A.1.4 Command Summary

The table A.1 summarizes the commands used within this section.

Table A.1: Summary of commands for MongoDB shell.

use	Create new database or returns existing one.
db.dropDatabase()	Drops an existing database.
show	Shows all current existing databases, collections or documents.
db.createCollection()	Creates a collection.
drop()	Deletes a specified collection.
insert()	Inserts data to a collection.
save()	Updates or inserts data to a collection.
find()	Queries data from a collection.
pretty()	Displays the results in a formatted way.
update()	Updates changes an existing document found by your find-parameters.
remove()	Removes an existing document.

A.2 Angular

A.2.1 Introduction to Angular

On the development of the front end of our project we are going to use Angular. Angular is a platform that makes it easy to build applications with the web. Angular combines declarative templates, dependency injection, end to end tooling, and integrated best practices to solve development challenges. Angular empowers developers to build applications that live on the web, mobile, or the desktop. [33]

A.2.2 Why Angular?

We choose angular because it fit our project needs. Multiple features like two-way binding and deep linking routing are both supported, the speed and performance is high and finally it is modular, this makes it easy to develop, understand and explain the code you are doing.

A.2.3 Installation of Angular

First of all we need to have NPM installed, we explained how to do so on: 4.3 Installation of NodeJS, section of this document.

We recommend to use Angular CLI. Angular CLI is a command line interface tool that can create a project, add files, and perform a variety of ongoing development tasks such as testing, bundling, and deployment. This will make the developing of the Angular application faster and easier.

To install Angular CLI globally we just need to type on the command line:

```
$ npm install -g @angular/cli
```

Finally to create a new project, navigate to the folder where you want to create it using a terminal window and then run the following command:

```
$ ng new my-app
```

This will create the project skeleton for your application and this will be its structure:

```
my-app
├── e2e
├── node_modules
├── src
│   ├── app
│   │   ├── app.component.css
│   │   ├── app.component.html
│   │   ├── app.component.spec.ts
│   │   ├── app.componnent.ts
│   │   └── app.module.ts
│   ├── assets
│   ├── environments
│   ├── favicon.ico
│   ├── index.html
│   ├── main.ts
│   ├── polyfills.ts
│   ├── styles.css
│   ├── test.ts
│   ├── tsconfig.app.json
│   ├── tsconfig.spec.json
│   └── typings.d.ts
├── .angular-cli.json
├── .editorconfig
├── karma.conf.js
└── package.json
```

```
├── package-lock.json
├── protractor.conf.js
├── README.md
├── tsconfig.json
└── tslint.json
```

To know more about angular their official web page is a nice option, it provides a "getting started" guide with how to install and use angular, a step by step tutorial on the main features angular provides such as the two way data binding, routing, implementing services, HTTP connections and many more. [34]

A.2.4 Angular basics

Modules

The modules are the building blocks that contain routes, components, services, directives, pipes, etc. All this blocks are related in a way that can be combined to form the angular application. To be able to define modules we have the decorator NgModule.

```
import { NgModule } from '@angular/core';

@NgModule({
  imports: [ ... ],
  declarations: [ ... ],
  bootstrap: [ ... ]
})
export class AppModule { }
```

Figure A.5: NgModule example.

In the example above, Figure: A.5, we have turned the class AppModule into an Angular module just by using the NgModule decorator. The NgModule decorator requires at least three properties: imports, declarations and bootstrap. The property imports expects an array of modules. Here's where we define the pieces of our puzzle (our application). The property declarations expects an array of components, directives and pipes that are part of the module. The bootstrap property is where we define the root component of our module. Even though this property is also an array, 99% of the time we are going to define only one component.

Directives

Directive modifies the DOM to change appearance, behaviour or layout of DOM elements. Directives are one of the core building blocks Angular uses to build applications. In fact, Angular components are in large part directives with templates. There are three main types of directives in Angular:

- Component - directive with a template.
- Attribute directives - directives that change the behaviour of a component or element but don't affect the template
- Structural directives - directives that change the behaviour of a component or element by affecting how the template is rendered

A service is a data layer, it contains the logic that is not component related, such as API requests.

Routing

When a user enters a web application or website, routing is their means of navigating throughout the application. To change from one view to another, the user clicks on the available links on a page.

Angular provides a Router to make it easier to define routes for the web applications and to navigate from one view to another view in the application.

The Routing renders a component based on the URL state. This feature of angular is extremely useful since with the tag <router-outlet> provided by angular, we can build a single page application quite easily just assigning the routes to each component and that component will render on the place of the <router-outlet>.

A.2.5 Single Page Applications (SPAs)

What is SPA?

A single-page application (SPA) is a web application or web site that interacts with the user by dynamically rewriting the current page rather than loading entire new pages from a server. This approach avoids interruption of the user experience between successive pages, making the application behave more like a desktop application. In an SPA, either all necessary code (HTML, JavaScript, and CSS) are retrieved with a single page load, or the appropriate resources are dynamically loaded and added to the page as necessary, usually in response to user actions. The page does not reload at any point in the process, nor does control transfer to another page, although the location hash or the HTML5 History API can be used to provide the perception and navigability of separate logical pages in the application. Interaction with the single page application often involves dynamic communication with the web server behind the scenes. [35]

Advantages of SPAs

Building our application as a SPA will give us a significant number of benefits:

- We will be able to bring a much better experience to the user
- The application will feel faster because less bandwidth is being used, and no full page refreshes are occurring as the user navigates through the application
- The application will be much easier to deploy in production, at least certainly the client part: all we need is a static server to serve a minimum of 3 files: our single page index.html, a CSS bundle, and a Javascript bundle.
- We can also split the bundles into multiple parts if needed using code splitting.
- The frontend part of the application is very simple to version in production, allowing for simplified deployment and rollbacks to previous version of the frontend if needed

And this just one possible deployment scenario of SPAs in production.

A.3 NodeJS

A.3.1 Introduction to NodeJS

To implement the back end of our application we used Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. Node.js' package ecosystem, NPM, is the largest ecosystem of open source libraries in the world. [36]

About NodeJS

Node is similar in design to, and influenced by, systems like Ruby's Event Machine or Python's Twisted. Node takes the event model a bit further. It presents an event loop as a runtime construct instead of as a library. In other systems there is always a blocking call to start the event-loop. Typically behaviour is defined through callbacks at the beginning of a script and at the end starts a server through a blocking call like EventMachine::run(). In Node there is no such start-the-event-loop call. Node

simply enters the event loop after executing the input script. Node exits the event loop when there are no more callbacks to perform. This behaviour is like browser JavaScript — the event loop is hidden from the user.

HTTP is a first class citizen in Node, designed with streaming and low latency in mind. This makes Node well suited for the foundation of a web library or framework.

Just because Node is designed without threads, doesn't mean you cannot take advantage of multiple cores in your environment. Child processes can be spawned by using our child_process.fork() API, and are designed to be easy to communicate with. Built upon that same interface is the cluster module, which allows you to share sockets between processes to enable load balancing over your cores. [37]

A.3.2 Why NodeJS?

Node.js enables developers to use Javascript on both the front and backend, which allows us to understand the entire stack and make changes if necessary. It is the perfect tool for developing high-throughput server-side applications. Node.js scales awlessly, which helps to save money in infrastructure costs, and since it is an open-source technology, it gives an edge with a shared repository of dynamic tools and modules that can be used instantly.

A.3.3 Installation of NodeJS

To install NodeJS, and with it NPM, you should follow the instructions on their web page, we recommend installing the most recent LTS to avoid most bugs and have the most support in case an error occurs. [38]

Appendix B

Packages

B.1 Back End

B.1.1 bcryptjs

This packages allows us to easily hash the passwords of the users that register on our web application using the bycrypt. It is a hashing algorithm which is scalable with hardware (via a configurable number of rounds). Its slowness and multiple rounds ensures that an attacker must deploy massive funds and hardware to be able to crack your passwords.[39]

Before we store the user credentials to our database we hash and salt the password in case some third person gained access to it. This way the password is protected so the attacker can not impersonate anyone on the application, the Figure:B.1 shows how it is implemented on the project.

```
bcrypt.genSalt(10, (err, salt) => {
  bcrypt.hash(newUser.password, salt, (err, hash) => {
    if (err) throw err;
    newUser.password = hash;
    newUser.save(callback);
  });
});
```

Figure B.1: bycript implementation to salt and hash password.

B.1.2 Express

Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications.It helps to organize your web application into an MVC architecture on the server side. To make a comparative Express.js is to Node.js what Ruby on Rails is to Ruby.

The API could be made without it but simplifies the code and callbacks in grate measure. [40] It provides us with an extensive API with all the methods to create our back end, from the routing system to specify what each end point does to a JWT validation.

On the Figure: B.2 we can see a simple example of a GET request on the route: 'localhost:3000/' that will return a classic "Hello World!".

```
var express = require('express');
var app = express();

app.get('/', function (req, res) {
  res.send('Hello World!');
});

app.listen(3000, function () {
  console.log('Example app listening on port 3000!');
});
```

Figure B.2: Express "Hello World"

Any other request (POST, PUT, DELETE...) or any other route will be responded with a 404 Not Found error message.

Middleware

Middleware functions are functions that have access to the request object (req), the response object (res), and the next function in the application's request-response cycle. The next function is a function in the Express router which, when invoked, executes the middleware succeeding the current middleware.

Middleware functions can perform the following tasks:

- Execute any code.
- Make changes to the request and the response objects.
- End the request-response cycle.
- Call the next middleware in the stack.

If the current middleware function does not end the request-response cycle, it must call next() to pass control to the next middleware function. Otherwise, the request will be left hanging. [41]

Some examples of express middleware packages used are:

- body-parser.

  Body parser is a package that allows us to parse incoming request bodies in a middleware before your handlers, available under the req.body property. This allows Express, to understand requests send in; application/x-www-form-urlencoded, multipart/form-data, application/json and many more. [42]

  On the project is used for urlencoded and JSON responses, the Figure: B.3 shows how easy it is to use.

```
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));
```

Figure B.3: Body-parser implementation.

- cookie-parser.

  Similar to body-parser this package allows us to parse Cookie header and populate req.cookies with an object keyed by the cookie names. Optionally we can enable signed cookie support by passing a secret string, which assigns req.secret so it may be used by other middleware. [43]

- cors.

  This package provides us with express middleware that we can use to enable CORS on our application with various options. [44]

B.1.3 jsonwebtoken

The jsonwebtoken package provides us with multiple functions that allows us to create, manipulate and verify the JWT on our back end, in this project is specially useful for it's verify function that allows us to check if the JWT we are receiving is indeed an authorized one and not a forged one. [45]

On the Figure: /reffigure:package:jwt-verify we can see an example of the use of the verify function.

```
jwt.verify(token, db.secretSession, (err, decoded) => {
  if (err) {
    return res.status(403).json({
      success: false,
      message: 'Failed to authenticate token'
    });
```

Figure B.4: JWT verify.

We can see if the verify is not positive it will return an error, if that occurs, the server will return a 403 Forbidden and deny the access.

B.1.4 MongoDB

Since we are going to use MongoDB as our data base, we will also be using the Node.js driver to connect it all together. The documentation about how to use it and it's API can be found on the following link. [46]

Mongoose

Mongoose is an Object Data Modeling (ODM) library for MongoDB and Node.js. It manages relationships between data, provides schema validation, and is used to translate between objects in code and the representation of those objects in MongoDB.[47]

Mongoose provides a straight-forward, schema-based solution to model your application data. It includes built-in type casting, validation, query building, business logic hooks and more, out of the box.[48] We can see an example of a mongoose model on the Figure: B.5

```
let UserSchema = mongoose.Schema({
  name: {
    type: String,
    required: true
  },
  username: {
    type: String,
    required: true,
    unique: true
  },
  password: {
    type: String,
    required: true
  },
  email: {
    type: String,
    required: true,
    unique: true
  }
});
```

Figure B.5: Mongoose object schema.

B.1.5 passport

Passport is authentication middleware for Node. It is designed to serve a singular purpose: authenticate requests. When writing modules, encapsulation is a virtue, so Passport delegates all other functionality to the application. This separation of concerns keeps code clean and maintainable, and makes Passport extremely easy to integrate into an application. [49]

B.1.6 passport-jwt

This package provides with a passport strategy for authenticating with a JSON Web Token. This module lets you authenticate endpoints using a JSON web token. It is intended to be used to secure RESTful endpoints without sessions. [50]

B.2 Front End

B.2.1 primeng

PrimeNG is a collection of rich UI components for Angular. All widgets are open source and free to use under MIT License.

This package allows us to add multiple components such as buttons, tables, alerts, calendars and much more to our Angular application easily without the need for us to create them. It also provides with multiple ways to include each component and examples on how to do it. [51]

B.2.2 font-awesome

Font Awesome is a font and icon toolkit based on CSS and LESS. It was made by Dave Gandy for use with Twitter Bootstrap, and later was incorporated into the BootstrapCDN. [52] We used font awesome as a source of icons for our front end application such as the once shown on the Figure: .

B.2.3 Bootstrap

Bootstrap is a free and open-source front-end library for designing websites and web applications. It contains HTML and CSS based designs templates for typography, forms, buttons, navigation and other interface components, as well as optional JavaScript extensions. Unlike many web frameworks, it concerns itself with front-end development only. [53]

We used some pre designed templates for our front end such as a navbar and a jumbotron.[54]. We are using bootstrap classes and grid system to style our HTML without need of our own custom CSS. [55]

B.2.4 ng2-pdf-viewer

This package provides a PDF Viewer Component for Angular. Since we will fill our front end with slides about various subjects and some of them are explained over PDF slides, we use this component to display them on our page. [56]

B.3 Development packages

B.3.1 Nodemon

Nodemon is a utility that will monitor for any changes in your source and automatically restart your server. We use this package so our server restarts when we save a change and we do not need to do it manually. [57]

Features:

- Automatic restarting of application.
- Default support for node & coffeescript, but easy to run any executable (such as python, make, etc).
- Ignoring specific files or directories.
- Watch specific directories.
- Works with server applications or one time run utilities and REPLs.
- Requirable in node apps.
- Open source and available on github.[58]

B.3.2 Angular-CLI

The Angular CLI makes it easy to create an application that already works, right out of the box. It already follows best practices. It also can generate components, routes, services and pipes with a simple command. The CLI will also create simple test shells for all of these. [59]

Appendix C

The Code

C.1 Back end

C.1.1 Main

Our main file is called "server.js" it holds the connection function to the database, it also contains a piece of middleware that checks for a CRSF token before going to the end points, this function will return a 403 if there is a token but it is not the correct one or a 401 if there is no token at all. This function will let pass the requests only on two cases; first if the token send on the request is valid, the other one is if the user is just logging in, accessing the /users/register or /users/login end points.

```
const https = require('https');
const http = require('http');
const fs = require('fs');

const express = require('express');
const bodyParser = require('body-parser');
const cors = require('cors');
const passport = require('passport');
const mongoose = require('mongoose');
const cookieParser = require('cookie-parser');
const jwt = require('jsonwebtoken');

// Configuration folder
const db = require('./config/db');

// HTTPS certifications
const keyFilePath = './config/privateKey.key';
const certFilePath = './config/certificate.crt';

// Ports
const httpPort = 8080;
const httpsPort = 8443;

const httpsOptions = {
  key: fs.readFileSync(keyFilePath, 'utf8'),
  cert: fs.readFileSync(certFilePath, 'utf8'),
  passphrase: 'Our not so secret passphrase'
};

const users = require('./routes/users');

const app = express();

// Middleware

// CORS
app.use(
  cors({
    credentials: true,
    origin: 'https://localhost:4200',
    methods: ['GET', 'PUT', 'POST']
  })
);

// Body Parser
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));
app.use(cookieParser());

// Passport
app.use(passport.initialize());
app.use(passport.session());

require('./config/passport')(passport);
```

```
app.use((req, res, next) => {
  const token = req.headers.authorization;
  if (req.url === '/users/login' || req.url === '/users/register') {
    next();
  } else if (token && req.cookies.jwt) {
    jwt.verify(token, db.secretSession, (err, decoded) => {
      if (err) {
        if (err.name === 'TokenExpiredError') {
          return res.status(403).json({
            success: false,
            message: 'Token expired please log in again',
            logout: true
          });
        } else {
          return res.status(403).json({
            success: false,
            message: 'Failed to authenticate token'
          });
        }
      } else {
        jwt.verify(req.cookies.jwt, db.secret, (err, decoded) => {
          if (err) {
            if (err.name === 'TokenExpiredError') {
              return res.status(403).json({
                success: false,
                message: 'Token expired please log in again',
                logout: true
              });
            } else {
              return res.status(403).json({
                success: false,
                message: 'Failed to authenticate token'
              });
            }
          } else {
            next();
          }
        });
      }
    });
  } else {
    res.status(401).json({ success: false, message: 'Unauthorized access' });
  }
});

// routes
app.use('/users', users);

mongoose.Promise = require('bluebird');

mongoose.connect(db.url, (err, database) => {
  if (err) return console.log(err);
  http.createServer(app).listen(httpPort);
  https.createServer(httpsOptions, app).listen(httpsPort);
});

mongoose.connection.on('connected', () => {
  console.log('Connected to database ' + db.url);
});

mongoose.connection.on('error', err => {
  console.log('Database error: ' + err);
});
```

Figure C.1: Back end "main".

C.1.2 Models

Inside the models folder we have our object model so it makes it easier to control and keep track of the parameters of it's object, what type they must be (string, number(integer) ...), it also allows us to set a required and unique flags for some parameters of the models, for example the email of a user.

It also contains the logic used in some functions of the "Routes" folder, such as "addUser" a functions that adds a new user to the database with the password salted and hashed to add a little bit of security if someone takes a pick on the database.

```
const mongoose = require('mongoose');
const bcrypt = require('bcryptjs');
const config = require('../config/db');

let UserSchema = mongoose.Schema({
  name: {
    type: String,
    required: true
  },
  username: {
    type: String,
    required: true,
    unique: true
  },
  password: {
    type: String,
    required: true
  },
  email: {
    type: String,
    required: true,
    unique: true
  }
});

const User = (module.exports = mongoose.model('User', UserSchema));

module.exports.getUserById = function(id, callback) {
  User.findById(id, callback);
};

module.exports.getUserByUsername = function(username, callback) {
  const query = { username: username };
  User.findOne(query, callback);
};

module.exports.addUser = function(newUser, callback) {
  bcrypt.genSalt(10, (err, salt) => {
    bcrypt.hash(newUser.password, salt, (err, hash) => {
      if (err) throw err;
      newUser.password = hash;
      newUser.save(callback);
    });
  });
};

module.exports.comparePassword = function(candidatePassword, hash, callback) {
  bcrypt.compare(candidatePassword, hash, (err, isMatch) => {
    if (err) throw err;
    callback(null, isMatch);
  });
};

module.exports.allUsers = function(callback) {
  User.find({}, callback);
};
```


Figure C.2: User model and queries.

C.1.3 Configuration

On the configuration folder there is our certificate and key to stablish the HTTPS connection. It also contains the db URL and the secrets to form the JWT. Finally we can see the passport configuration file which is used to check if the token inside the cookie is valid or not.

```
const JwtStrategy = require('passport-jwt').Strategy;
const ExtractJwt = require('passport-jwt').ExtractJwt;
const User = require('../models/users');
const config = require('../config/db');

let cookieExtractor = function(req) {
  let token = null;
  if (req && req.cookies) {
    token = req.cookies['jwt'];
  }
  return token;
};

module.exports = function(passport) {
  let opts = {};
  opts.jwtFromRequest = cookieExtractor;
  opts.secretOrKey = config.secret;

  passport.use(
    new JwtStrategy(opts, (jwt_payload, done) => {
      User.getUserById(jwt_payload._id, (err, user) => {
        if (err) {
          return done(err, false);
        }
        if (user) {
          return done(null, user);
        } else {
          return done(null, false);
        }
      });
    })
  );
};
```

Figure C.3: Passport configuration.

C.1.4 End Points

Routes inside /users

Inside our "/users" route we have multiple routes to access and store data.

All the responses will come with a messages with extra information to let the client know what went wrong.

/register:

- POST: Used to store new users to the database. This route will return a 409 if the user is duplicated, a 500 if there is an unexpected error or a 200 otherwise.

/login:

- POST: Used to check if the username/password combination are a match for a user stored on the database. This route will return a 404 if the user is not found, a 400 if the user is found but the password is wrong, a 500 if there is an unexpected error or a 200 otherwise.

On this route we will set up the tokens to avoid CSRF & XSS attaks. To do so, one token will be send as a set-cookie with the flags "HTTPOnly" and "Secure", the other will be send on the response so the client can store it.

/profile:

-GET: Used to retrieve some of the user information. This route is protected, the cookie send on the "/login" must match the one send on this petition otherwise the server will return an unauthorized response.

```
const express = require('express');
const router = express.Router();
const passport = require('passport');
const jwt = require('jsonwebtoken');
const config = require('../config/db');
const User = require('../models/users');

const ObjectID = require('mongodb').ObjectID;

router.post('/register', (req, res) => {
  const user = req.body;
  let newUser = new User({
    name: user.name,
    email: user.email,
    username: user.username,
    password: user.password
  });

  User.addUser(newUser, (err, user) => {
    if (err) {
      if (err['code'] === 11000) {
        res.status(409).json({
          success: false,
          message: 'Failed to register user, duplicated'
        });
      } else {
        res.status(500).json({
          success: false,
          message: 'Failed to register user',
          error: err['message']
        });
      }
    } else {
      res.json({ success: true, message: 'User registered' });
    }
  });
});
```

```
router.post('/login', (req, res) => {
  const username = req.body.username;
  const password = req.body.password;
  User.getUserByUsername(username, (err, user) => {
    if (err) throw err;
    if (!user) {
      return res
        .status(404)
        .json({ success: false, message: 'User not found' });
    }
    User.comparePassword(password, user.password, (err, isMatch) => {
      if (err) throw err;
      if (isMatch) {
        const token = jwt.sign(user.toJSON(), config.secret, {
          expiresIn: 21600 // 6h
        });
        const csrfToken = jwt.sign(user.toJSON(), config.secretSession, {
          expiresIn: 21600 // 6h
        });
        res.cookie('jwt', token, {
          expires: new Date(Date.now() + 21600000), // 6h
          secure: true,
          httpOnly: true
        });
        res.json({
          success: true,
          message: 'Logged in',
          csrfToken: csrfToken
        });
      } else {
        return res
          .status(400)
          .json({ success: false, message: 'Wrong password' });
      }
    });
  });
});

router.get(
  '/profile',
  passport.authenticate('jwt', { session: false }),
  (req, res) => {
    res.json({
      name: req.user.name,
      username: req.user.username,
      email: req.user.email
    });
  }
);

module.exports = router;
```

Figure C.4: End point functions.

C.2 Front end

C.2.1 Components

Components are the most basic building block of an UI in an Angular application. An Angular application is a tree of Angular components.

Home

The home component provides the user with a summary of what the page contains and a button to log in the application, this button will redirect the user to the same page as the login button of the top right corner of the page, on the navigation bar. This button will not appear if the user is already logged inside the application.

```
import { Component, OnInit } from '@angular/core';
import { Router } from '@angular/router';

// services
import { AuthService } from '../../core/services/auth.service';

@Component({
  selector: 'app-home',
  templateUrl: './home.component.html',
  styleUrls: ['./home.component.css']
})
export class HomeComponent implements OnInit {
  constructor(private router: Router, protected authService: AuthService) {}

  ngOnInit() {}

  learnMore() {
    this.router.navigate(['./login']);
  }
}
```

Figure C.5: home.component.ts

```
<div class="jumbotron">
  <h1 class="display-3">{{ 'home-title' | translate }}</h1>
  <p class="lead">{{ 'home-subtitle' | translate }}</p>
  <hr class="my-4">
  <p>{{ 'home-subtitle-2' | translate }}</p>
  <p class="lead">
    <a *ngIf="!authService.loggedIn()" class="btn btn-primary btn-lg text-light" (click)="learnMore()" role="button">{{'login' | translate}}</a>
  </p>
</div>
```

Figure C.6: home.component.html

Login

The login component contains a basic login form with two boxes to input text, user & password, and a button to confirm the user finished typing on the fields.

It contains a multiple validators. On one hand, to make sure the user fills all the fields, it checks if all the fills have been typed on, takes place fully on the front end. On the other had to validate the user/password combination the page makes a

request to the back end to do it. If an error occurs, a pop up will show to inform the user that an error happened and will provide some information about it.

```typescript
import { Component, OnInit } from '@angular/core';
import {
  FormGroup,
  FormControl,
  Validators,
  FormBuilder
} from '@angular/forms';
import { Router } from '@angular/router';

// services
import { AuthService } from '../../core/services/auth.service';

// Models
import { UserResponse } from '../../core/models/userResponse.model';

// primeng
import { GrowlModule } from 'primeng/components/growl/growl';
import { Message } from 'primeng/components/common/api';

@Component({
  selector: 'app-login',
  templateUrl: './login.component.html',
  styleUrls: ['./login.component.css']
})
export class LoginComponent implements OnInit {
  loginForm: FormGroup;
  msg: Message[] = [];
  constructor(
    private formBuilder: FormBuilder,
    private authService: AuthService,
    private router: Router
  ) {}

  ngOnInit() {
    this.loginForm = this.formBuilder.group({
      username: ['', Validators.required],
      password: ['', Validators.required]
    });
  }

  onLoginSubmit() {
    if (!this.loginForm.valid) {
      this.createMessage('Form Not Valid: Please fill all the fields.', false);
    } else {
      this.authService.authenticateUser(this.loginForm.value).subscribe(
        (data: UserResponse) => {
          this.createMessage(data.message, data.success);
          if (data.success) {
            sessionStorage.setItem('idToken', data['csrfToken']);
            this.router.navigate(['/profile']);
          }
        },
        error => {
          this.createMessage(error.error.message, error.error.success);
        }
      );
    }
  }
  createMessage(message: string, isCorrect: boolean) {
    this.msg = [];
    if (isCorrect) {
      this.msg.push({
        severity: 'success',
        summary: 'Success Message',
        detail: message
      });
    } else {
      this.msg.push({
        severity: 'error',
        summary: 'Error Message',
        detail: message
      });
    }
  }
}
```

Figure C.7: login.component.ts

```
<p-growl [(value)]="msg"></p-growl>
<h2 class="page-header" >{{'login'|translate}}</h2>
<form [formGroup]="loginForm" (submit)="onLoginSubmit()">
  <div class="form-group">
    <label>{{'username'|translate}}</label>
    <input type="text" formControlName="username" class="form-control">
  </div>
  <div class="form-group">
    <label>{{'password'|translate}}</label>
    <input type="password" formControlName="password" class="form-control">
  </div>
  <button type="submit" class="btn btn-primary">{{'login'|translate}}</button>
</form>
```

Figure C.8: login.component.html

Navbar

The navbar component is a navigation bar on the top of the page that allows the user to navigate through the multiple pages.

The pages which the navigation bar displays depends if the user is logged in or not. This check is done by the "authService" a service that we will discuss later.

```ts
import { Component, OnInit } from '@angular/core';
import { Router } from '@angular/router';

// services
import { AuthService } from '../../core/services/auth.service';

// primeng
import { SidebarModule } from 'primeng/sidebar';

@Component({
  selector: 'app-navbar',
  templateUrl: './navbar.component.html',
  styleUrls: ['./navbar.component.css']
})
export class NavbarComponent implements OnInit {
  display: boolean;
  constructor(protected authService: AuthService, private router: Router) {}

  ngOnInit() {
    this.display = false;
  }

  onLogout() {
    sessionStorage.clear();
  }

  goTcgiCourse() {
    this.router.navigate(['courses/tcgi']);
  }

  goSecurityCourse() {
    this.router.navigate(['courses/security']);
  }
}
```

Figure C.9: navbar.component.ts

```
<nav class="navbar navbar-expand-lg navbar-light bg-dark">
  <button *ngIf="authService.loggedIn()" class="btn btn-dark" (click)="display = true">
    <span class="glyphicon glyphicon-th-list"></span> 
  </button>
  <button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#navbarSupportedContent" aria-controls="navbarSupportedContent"
    aria-expanded="false" aria-label="Toggle navigation">
    <span class="navbar-toggler-icon"></span>
  </button>
  <div class="collapse navbar-collapse" id="navbarSupportedContent">
    <ul class="navbar-nav mr-auto">
      <li class="nav-item" [routerLinkActive]="['active']" [routerLinkActiveOptions]="{exact:true}">
        <a class="nav-link text-light" [routerLink]="['/home']">{{'home'|translate}}</a>
      </li>
      <li class="nav-item ml-3 mt-2">
        <div class="row mt-1">
          <div class="col">
            <div ngbDropdown *ngIf="authService.loggedIn()" class="d-inline-block">
              <button class="btn btn-outline-primary text-light" id="dropdownBasic1" ngbDropdownToggle>{{'course-selection' | translate }}</button>
              <div ngbDropdownMenu aria-labelledby="dropdownBasic1">
                <button class="dropdown-item" (click)="goTcgiCourse()">TCGI</button>
                <button class="dropdown-item" (click)="goSecurityCourse()">{{'security-basics' | translate}}</button>
              </div>
            </div>
          </div>
        </div>
      </li>
    </ul>
    <ul class="nav navbar-nav navbar-right">
      <li class="nav-item" *ngIf="authService.loggedIn()" [routerLinkActive]="['active']" [routerLinkActiveOptions]="{exact:true}">
        <a class="text-light" [routerLink]="['/profile']">{{'profile'|translate}}</a>
      </li>
      <li class="nav-item" *ngIf="!authService.loggedIn()" [routerLinkActive]="['active']" [routerLinkActiveOptions]="{exact:true}">
        <a class="text-light" [routerLink]="['/login']">{{'login'|translate}}</a>
      </li>
      <li class="nav-item" *ngIf="!authService.loggedIn()" [routerLinkActive]="['active']" [routerLinkActiveOptions]="{exact:true}">
        <a class="text-light" [routerLink]="['/register']">{{'register'|translate}}</a>
      </li>
      <li class="nav-item" *ngIf="authService.loggedIn()" [routerLinkActive]="['active']" [routerLinkActiveOptions]="{exact:true}">
        <a class="text-light" (click)="onLogout()" [routerLink]="['/login']">{{'logout'|translate}}</a>
      </li>
    </ul>
  </div>
</nav>
<p-sidebar [(visible)]="display">
  Content
</p-sidebar>
```

Figure C.10: navbar.component.html

```
.active a {
  background: black !important;
}

a.nav-link {
  padding: 1.5rem !important;
}

button:hover {
  background-color: transparent;
  border: transparent;
}
```

Figure C.11: navbar.component.css

Profile

The profile component shows some basic information provided by the user on register; the name and email of the logged user. The profile component asks for this information to the back end using the unique token to identify the user.

```
import { Component, OnInit } from '@angular/core';
import { Router } from '@angular/router';

// models
import { User } from '../../core/models/user.model';

// services
import { AuthService } from '../../core/services/auth.service';
import { UserResponse } from '../../core/models/userResponse.model';

@Component({
  selector: 'app-profile',
  templateUrl: './profile.component.html',
  styleUrls: ['./profile.component.css']
})
export class ProfileComponent implements OnInit {
  user: User;

  constructor(private authService: AuthService, private router: Router) {}

  ngOnInit() {
    this.authService.getProfile().subscribe((data: User) => {
      this.user = data;
    });
  }
}
```

Figure C.12: profile.component.ts

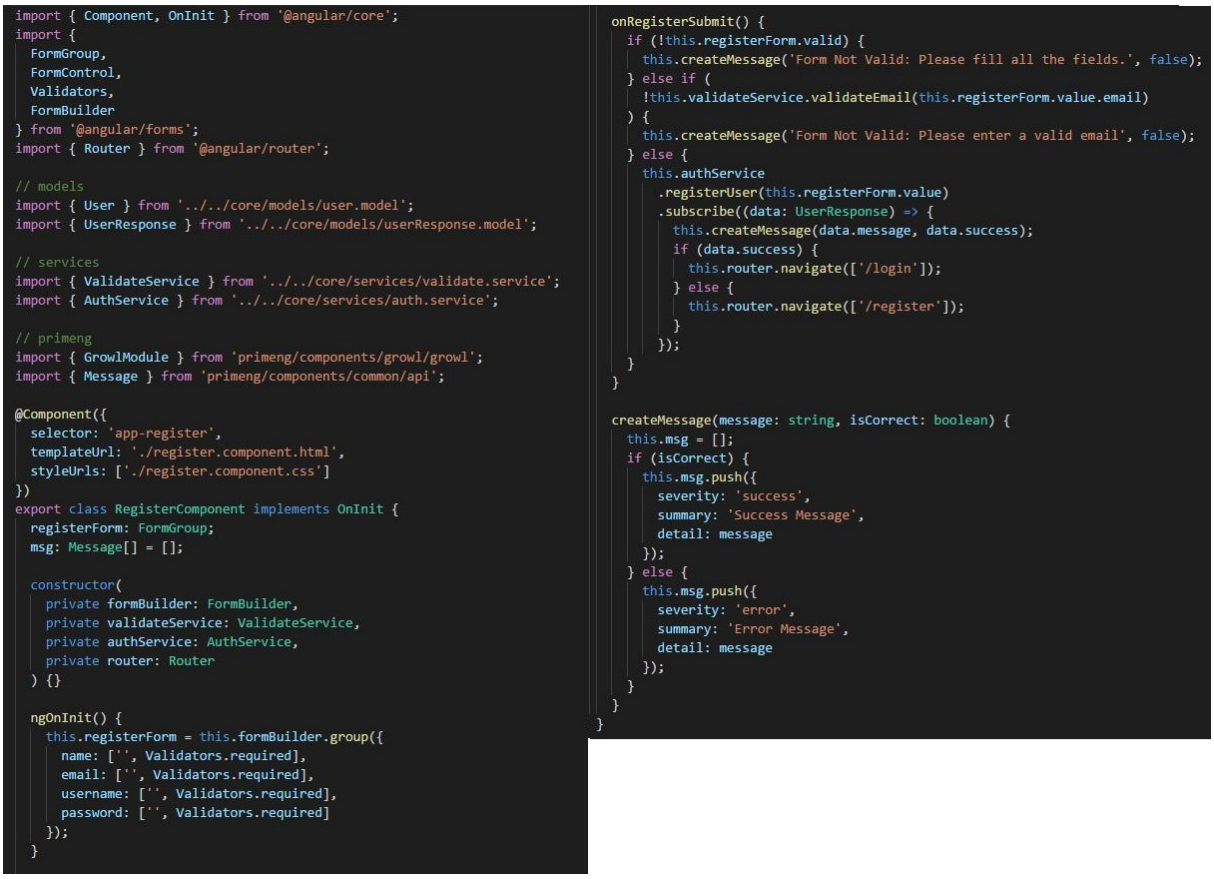
```
<div *ngIf="user">
  <h2 class="page-header">{{user['name']}}</h2>
  <ul class="list-group">
    <li class="list-group-item">Username: {{user.username}}</li>
    <li class="list-group-item">email: {{user.email}}</li>
  </ul>
</div>
```

Figure C.13: profile.component.html

Register

The register component contains a basic login form with four boxes to input text, user, username, password and email; and a button to confirm the user finished typing on the fields.

It contains a multiple validators. First it checks if all the fields are full, then it test if the email is possible, checking for the form: XXX@XXX.X, both this verifications are done exclusively on the front end. Finally it checks if the user and the password are unique, this is done on the back end. If an error occurs, a pop up will show to inform the user that an error happened and will provide some information about it.



```
import { Component, OnInit } from '@angular/core';
import {
  FormGroup,
  FormControl,
  Validators,
  FormBuilder
} from '@angular/forms';
import { Router } from '@angular/router';

// models
import { User } from '../../core/models/user.model';
import { UserResponse } from '../../core/models/userResponse.model';

// services
import { ValidateService } from '../../core/services/validate.service';
import { AuthService } from '../../core/services/auth.service';

// primeng
import { GrowlModule } from 'primeng/components/growl/growl';
import { Message } from 'primeng/components/common/api';

@Component({
  selector: 'app-register',
  templateUrl: './register.component.html',
  styleUrls: ['./register.component.css']
})
export class RegisterComponent implements OnInit {
  registerForm: FormGroup;
  msg: Message[] = [];

  constructor(
    private formBuilder: FormBuilder,
    private validateService: ValidateService,
    private authService: AuthService,
    private router: Router
  ) {}

  ngOnInit() {
    this.registerForm = this.formBuilder.group({
      name: ['', Validators.required],
      email: ['', Validators.required],
      username: ['', Validators.required],
      password: ['', Validators.required]
    });
  }

  onRegisterSubmit() {
    if (!this.registerForm.valid) {
      this.createMessage('Form Not Valid: Please fill all the fields.', false);
    } else if (
      !this.validateService.validateEmail(this.registerForm.value.email)
    ) {
      this.createMessage('Form Not Valid: Please enter a valid email', false);
    } else {
      this.authService
        .registerUser(this.registerForm.value)
        .subscribe((data: UserResponse) => {
          this.createMessage(data.message, data.success);
          if (data.success) {
            this.router.navigate(['/login']);
          } else {
            this.router.navigate(['/register']);
          }
        });
    }
  }

  createMessage(message: string, isCorrect: boolean) {
    this.msg = [];
    if (isCorrect) {
      this.msg.push({
        severity: 'success',
        summary: 'Success Message',
        detail: message
      });
    } else {
      this.msg.push({
        severity: 'error',
        summary: 'Error Message',
        detail: message
      });
    }
  }
}
```

Figure C.14: register.component.ts

```
<p-growl [(value)]="msg"></p-growl>
<h2 class="page-header" >{{'register'|translate}}</h2>
<form [formGroup]="registerForm" (submit)="onRegisterSubmit()">
  <div class="form-group">
    <label>{{'name'|translate}}</label>
    <input type="text" formControlName="name" class="form-control">
  </div>
  <div class="form-group">
    <label>{{'username'|translate}}</label>
    <input type="text" formControlName="username" class="form-control">
  </div>
  <div class="form-group">
    <label>{{'email'|translate}}</label>
    <input type="text" formControlName="email" class="form-control">
  </div>
  <div class="form-group">
    <label>{{'password'|translate}}</label>
    <input type="password" formControlName="password" class="form-control">
  </div>
  <button type="submit" class="btn btn-primary">{{'register'|translate}}</button>
</form>
```

Figure C.15: register.component.html

C.2.2 core

Models

The models are Objects created by us to make possible to add a custom types of variables, this is a feature of Typescript that we are taking advantage of.

```
export class User {
  name: string;
  email: string;
  username: string;
}
```

```
export class UserResponse {
  success: boolean;
  message: string;
  token?: string;
  user?: Object;
}
```

Figure C.16: user.model.ts & userResponse.model.ts

Services

An angular service is simply a function that allows you to access its' defined properties and methods. It also helps keep your coding organized.

- authService: The authentication service is the responsible of the exchange of information about the user and handle the data send from the back end so the components can display it or it is stored. It uses a HttpClient object form angular to do the request to the server.

registerUser(user): This function is called when the user clicks on the button of the register and the form passes the front end validators. This does a POST request to the back end with the information provided on the register form.

authenticateUser(user): This function is called when the user clicks on the button of the login and the form passes the front end validators. This does a POST request to the back end with the information provided on the register form.

getProfile(): This function does a GET request to the back end with the security tokens of the user, this request will return information about the user matching the tokens.

loggedIn(): This function checks if the user is logged in checking the sessionStorage for a token.

```typescript
import { Injectable } from '@angular/core';
import { HttpClient, HttpHeaders } from '@angular/common/http';
import { Observable } from 'rxjs/Observable';

// model
import { UserResponse } from '../../core/models/userResponse.model';
import { User } from '../models/user.model';

@Injectable()
export class AuthService {
  baseURL = 'https://localhost:8443';
  constructor(private http: HttpClient) {}

  registerUser(user) {
    const headers: HttpHeaders = new HttpHeaders();
    headers.append('Content-Type', 'application/json');
    return this.http
      .post(this.baseURL + '/users/register', user, { headers: headers })
      .map(res => res);
  }

  authenticateUser(user) {
    const headers = new HttpHeaders();
    headers.append('Content-Type', 'application/json');
    return this.http
      .post(this.baseURL + '/users/login', user, {
        headers: headers,
        withCredentials: true
      })
      .map(res => {
        return res;
      });
  }

  getProfile() {
    const headers = new HttpHeaders({
      'Content-Type': 'application/json'
    });
    return this.http
      .get(this.baseURL + '/users/profile', {
        headers: headers,
        withCredentials: true
      })
      .map((res: User) => {
        return res;
      });
  }

  loggedIn() {
    if (sessionStorage.getItem('idToken')) {
      return true;
    }
    return false;
  }
}
```

Figure C.17: auth.service.ts

C.2.3 Shared

Creating shared modules allows you to organize and streamline your code. You can put commonly used directives, pipes, and components into one module and then import just that module wherever you need it in other parts of your app.

In our project we are using a translate pipe that allows us to, after createing multiple JSONs for multiple languages with the same key and the value is different for each one, we can change between them to make the page multi lenguage.

Guards

The guards are the component that protects the routes when a user is not logged in to prevent the access, and redirects to home page if the user is not authorized to access the page the URL is taking him to.

```typescript
import { Injectable } from '@angular/core';
import { Router, CanActivate } from '@angular/router';

// service
import { AuthService } from '../core/services/auth.service';

@Injectable()
export class AuthGuard implements CanActivate {
  constructor(private authService: AuthService, private router: Router) {}
  canActivate() {
    if (this.authService.loggedIn()) {
      return true;
    } else {
      this.router.navigate(['/home']);
      return false;
    }
  }
}
```

Figure C.18: auth.guard.ts

Assets

The assets folder contains all the JSON for the translate pipe and the images used for the UI.

```json
{
  "home": "Home",
  "profile": "Profile",
  "login": "Login",
  "register": "Register",
  "name": "Name",
  "username": "Username",
  "email": "email",
  "password": "Password",
  "logout": "Logout",
  "marks": "Marks",
  "home-title": "Welcome to my final degree project",
  "home-subtitle": "The objective of this page is to test the security implementations studied within the project.",
  "home-subtitle-2": "This applicaiton is a PoC",
  "security-basics": "Security Basics",
  "course-selection":"Select a course"
}
```

Figure C.19: en.json

Appendix D

Work plan

Web Authorization and authentication for single page applications (SPAs)	WP ref: P1
Major constituent: Software	
Short description: Preparation and installation of all the software needed.	Start date: 10/01/2018 End date: 20/01/2018
Internal task T1: Installation of MongoDB. Internal task T2: Installation of NodeJS and all the required packages (Node Modules). Internal task T3: Installation of Angular.	

Web Authorization and authentication for single page applications (SPAs)	WP ref: P2
Major constituent: Learning	
Short description: Familiarization with the languages and Main protocols & security threats.	Planned start date: 20/01/2018 Planned end date: 27/05/2018
Internal task T1: Study of MongoDB and mongo shell. Internal task T2: Study of expressJS and NodeJS. Internal task T3: Study of Angular. Internal task T4: Study of security protocols and technologies HTTP, HTTPS, web token, JWT, CORS, sessionID. Internal task T5: Research and study of security threads.	

Web Authorization and authentication for single page applications (SPAs)	WP ref: P3
Major constituent: Programming & Documentation	
Short description: Program a working back end with a mongoDB running locally and document it.	Planned start date: 20/02/2018 Planned end date: 07/03/2018
Internal task T1: Create the back end structure. Internal task T2: Creation of the end points with the main calls used (get, put, post, delete). Internal task T3: Test with postman/curl the correct functioning of this end points. Internal task T4: Documentation of the installation of the technologies used on this package and how to use them.	

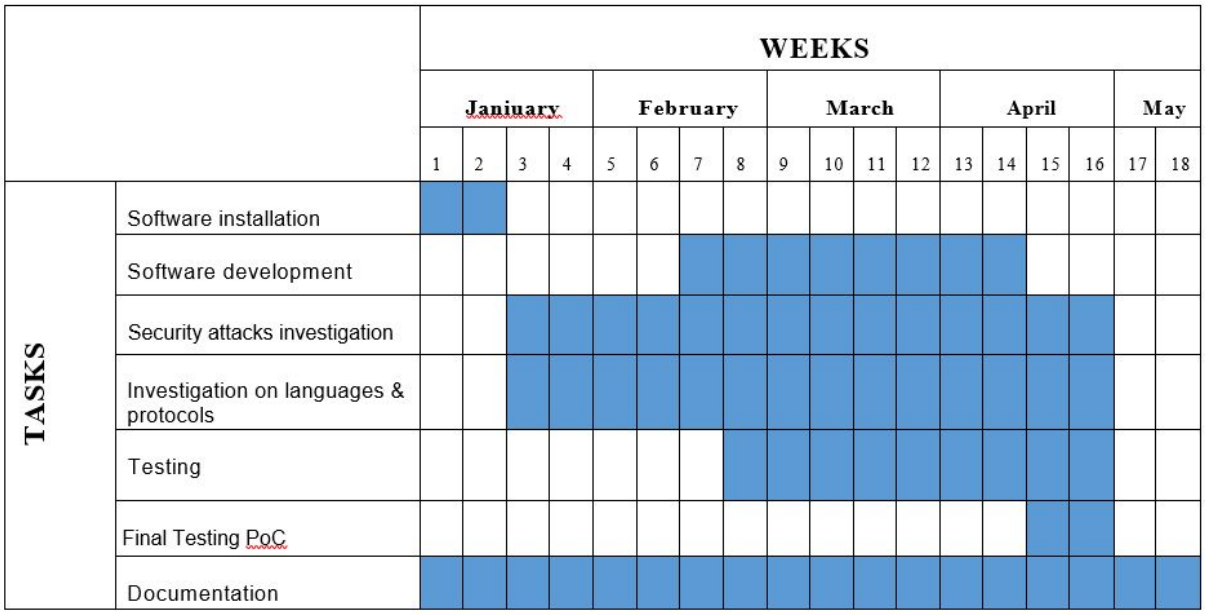
Web Authorization and authentication for single page applications (SPAs)	WP ref: P4
Major constituent: Programming & Documentation	
Short description: Program a working front end with Angular and document it.	Planned start date: 07/03/2018 Planned end date: 05/04/2018
Internal task T1: Create the front-end structure. Internal task T2: Creation of main views and the workflow of the page. Internal task T3: Test the main views and the workflow of the page. Internal task T4: Documentation of the installation of the technologies used on this package and how to use them.	

Web Authorization and authentication for single page applications (SPAs)	WP ref: P5
Major constituent: PoC & Documentation	
Short description: Merge both P3 and P4 and test the PoC.	Planned start date: 06/04/2018 Planned end date: 01/05/2018
Internal task T1: Connect the backend end points with the front end. Internal task T2: Match all the object calls and modifications if needed. Internal task T3: Testing of the full stack application.	

Web Authorization and authentication for single page applications (SPAs)	WP ref: P6
Major constituent: Documentation	
Short description: Write all the documentation mandatory for the UPC	Planned start date: 15/01/2018 Planned end date: 11/05/2018
Internal task T1: Realization of project proposal and workplan. Internal task T2: Realization of critical review. Internal task T3: Realization of TFG memory. Internal task T4: Preparation of TFG presentation.	

Milestones

WP#	Task#	Short title	Milestone / deliverable	Date (week)
P1	T3	Software installation	Left the local machine set and ready to start the project	2
P2	T4	Learning the language	Knowledge about the languages & protocols to be used	4
P3	T3	Back end programming	Back end running and tested	6
P3	T4	Back end documentation	Back end documentation done	6
P4	T3	Front end programming	Front end working as wanted with a good look and feel.	12
P4	T4	Front end documentation	Front end documentation done	12
P5	T3	Final testing of PoC	The PoC of the web application working as expected.	16
P5	T4	TFG memory	End of TFG documentation	17
P5	T5	TFG presentation	End of TFG	19



Glossary

XSS - Cross Site Scripting
CSRF - Cross Site Request Forgery
MITM - Man In The Middle
SPA - Single Page Application
CA - Certificate Authority
NPM - Node Package Manager
DOM - Document Object Model
PoC - Proof of Concept.
HTTP - Hypertext Transfer Protocol.
UPD - User Datagram Protocol.
API - Application Programming Interface.
TLS - Transport Layer Security.
SSL - Secure Sockets Layer.
SOP - Same-Origin policy.
CORS - Cross-Origin Resource Sharing.