
Security First approach in development of Single-Page Application based on Angular

UNIVERSITY OF TURKU
DEPARTMENT OF FUTURE TECHNOLOGIES
MASTER OF SCIENCE IN TECHNOLOGY THESIS
SECURITY OF NETWORKED SYSTEMS
September 2019
Daniel Danielecki

Supervisors:
Sampsa Rauti (University of Turku)
Dr Seppo Virtanen (University of Turku)
Thomas Beekman (KPMG N.V.)

The originality of this thesis has been checked in accordance with the University of Turku quality assurance system using the Turnitin Originality Check service.

UNIVERSITY OF TURKU

Department of Future Technologies

DANIEL DANIELECKI: Security First approach in development of Single-Page Application based on Angular

Master of Science in Technology Thesis, 70 p., 8 app. p.

Security of Networked Systems

September 2019

Recently a Single-Page Application (SPA) approach is getting attention even though this is based on JavaScript is not considered to be a safe programming language. In the SPA ecosystem developers often have to use many external dependencies. Detected vulnerabilities in these external dependencies are disclosed and updated in most cases by the community. Often, in-depth security analysis is not included during the development stage, due to project deadlines and other circumstances. It goes with number of complications. The most straightforward is to be vulnerable for cyber attacks which causes financial problems for companies. Currently law already includes penalties in case of data breaches. Moreover, detected vulnerable code delays projects due to necessary time to improve it. Sometimes it requires to change the whole architecture if the application was poorly designed or in case security was skipped completely in the early stage. It might lead even to putting changes in the architectural style once the application is already on the market. It does makes high pressure on software developers to fix it fast. The rush to deliver it as fast as possible can create new security risks, because in some scenarios it might take significant amount of time to change the design with security prioritization.

Especially within the financial industry consequences of not including security during the design stage might be harmful. Companies in this industry are entrusted with high social trust and sensitive (personal) data. For such enterprises shortcomings in security might cause data, image and money loss. Cybercrime activities are intensifying and for some companies it might causes to be kicked out of business due to hacking. This important factor of software development is currently getting more attention. That is why providing security in an early stage of a project is important, as well should be considered as a prerequisite.

Security should be integrally included in all parts of the development cycle: specification, design, implementation and testing. The desired result is a secure web application. Improving security might be done explicitly by using security analysis and enhance security accordingly to the results. However, implicit methods like clean code, programming best practices, proper architecture design also applies. Ideally, in a continuous security way. Programming best practices and countermeasures against web application security threats have been used to analyse and verify SPA security.

In this research project, an Angular SPA has been developed with focus on security. It includes programming best practices, security analysis and number of different tests. The main goal was to develop a SPA based on the Angular framework with security first approach. An in-depth security analysis of the deployed application is then conducted with validation of these results.

Keywords: Angular, continuous security, JavaScript, programming best practices, Single-Page Application, web security.

Acknowledgments

The thesis was a final project for Master Degree at University of Turku (Finland) on Cyber Security track with Networked Systems Security specialization. This was an exit university, one of two universities for EIT Digital Master School. This was a double degree program performed by author of this thesis, where the entry university was University of Twente (Netherlands). Master of Science in Technology Thesis is written at University of Turku, the supervisors from university side are Sampsa Rauti, together with Dr Seppo Virtanen.

Performed internship at KPMG in their headquarters in Amstelveen (Netherlands) provided help during this research. KPMG is a global consulting company specialized in accountancy, advisory and tax services, which was originally founded in 1818. The Dutch firm belonging to the KPMG has legal name KPMG N.V. Their focus is on advisory, financial audit and tax. Within their broad portfolio of IT services, KPMG N.V. has a strong track record in cyber security and software development. This thesis combines both of those two areas. The Digital Enablement department of KPMG N.V. is the software development related division. This is the department, in which the thesis has been co-created, especially from the practical side. Many thanks for Thomas Beekman. He supervised the thesis from the business side.

The purpose of this research is to study security in Angular-based SPA. Therefore, all information provided here, i.e. security investigation is for educational purposes only. Should not be used for any other intention than study of this topic. No one involved in this project, including author of the thesis, is responsible for possible damages it can cause, if not used correctly or ethically.

Contents

| | |
|--|-----------|
| List of Figures | iv |
| List of Tables | v |
| Listings | vi |
| 1 Introduction | 1 |
| 1.1 Background | 1 |
| 1.2 Problem | 1 |
| 1.3 Goal | 2 |
| 1.4 Methodology | 2 |
| 1.5 Scope | 3 |
| 1.6 Thesis Structure | 3 |
| 2 Architecture and Security | 4 |
| 2.1 Software Architecture | 4 |
| 2.1.1 Client-Server | 4 |
| 2.1.2 Multitier architecture | 5 |
| 2.1.3 Service-Oriented Architecture | 6 |
| 2.1.4 Model-View-Controller | 6 |
| 2.1.5 Model-View-ViewModel | 7 |
| 2.1.6 Component-Based Architecture | 8 |
| 2.2 Software Security | 9 |
| 2.2.1 Risks | 11 |
| 2.2.2 Vulnerabilities | 13 |
| 2.2.3 Threats | 13 |
| 2.2.4 Control | 14 |
| 2.2.5 Secure Software Development Life Cycle | 14 |
| 2.2.6 SecDevOps | 15 |
| 3 Single-Page Applications | 17 |
| 3.1 Web Applications | 17 |
| 3.1.1 JavaScript | 17 |
| 3.1.2 TypeScript | 19 |
| 3.2 Frameworks | 19 |
| 3.3 Angular | 19 |
| 3.3.1 Security | 20 |
| 3.3.2 Programming Best Practices | 24 |
| 3.3.3 Testing | 26 |
| 3.4 Workflow | 27 |
| 4 Design | 28 |
| 4.1 Application Architecture | 28 |
| 4.1.1 Modules | 29 |
| 4.1.2 Components | 31 |
| 4.1.3 State Management | 32 |
| 4.2 Application Security | 32 |
| 4.2.1 Containerization | 32 |
| 4.2.2 Compilers Rules | 34 |
| 4.2.3 Secure Development | 36 |

| | | |
|----------|--|-----------|
| 4.2.4 | Security Testing | 38 |
| 4.2.5 | Continuous Security | 39 |
| 4.3 | Behavior-Driven Development | 40 |
| 5 | Implementation | 41 |
| 5.1 | Functionalities | 41 |
| 5.2 | Technology Stack | 42 |
| 5.3 | Improving Security | 43 |
| 5.3.1 | Input Validation | 44 |
| 5.3.2 | File Upload Attack Prevention | 44 |
| 5.3.3 | Security Headers | 45 |
| 5.3.4 | Content Security Policy | 47 |
| 5.3.5 | Others | 49 |
| 5.4 | Clean Code | 50 |
| 5.5 | Unit Testing | 52 |
| 5.6 | Server-Side Rendering | 53 |
| 5.7 | Version Control System | 54 |
| 5.8 | Deployment | 54 |
| 5.9 | Transpilation | 55 |
| 6 | Security Analysis | 56 |
| 6.1 | Static Application Security Testing | 56 |
| 6.2 | Dynamic Application Security Testing | 57 |
| 6.2.1 | Information Gathering | 57 |
| 6.2.2 | Web Scanners | 59 |
| 6.2.3 | Skipfish | 61 |
| 6.2.4 | Stress Testing | 61 |
| 6.2.5 | Others | 62 |
| 6.3 | Gray Box Testing | 63 |
| 6.3.1 | Cross-Site Request Forgery | 63 |
| 6.3.2 | File Upload Attack | 63 |
| 6.3.3 | API Keys | 63 |
| 6.4 | Results | 64 |
| 7 | Conclusion | 65 |
| | References | 66 |
| | Appendices | 71 |
| A | UI Design Elements | 71 |
| | Index | 74 |

List of Figures

| | | |
|----|---|----|
| 1 | Client-server architecture in the Internet. | 5 |
| 2 | Multitier architecture, often present in modern applications. | 6 |
| 3 | MVC architectural style. | 7 |
| 4 | MVVM architectural pattern. | 8 |
| 5 | Component-based architectural pattern. | 9 |
| 6 | Seven software security touchpoints [15]. | 10 |
| 7 | SDLC methodology. | 15 |
| 8 | SSDLC methodology. | 15 |
| 9 | Logical layers separation of a typical web application. | 17 |
| 10 | Most popular technologies, annual report by Stack Overflow for 2018 [26]. Includes languages with at least 10% usage by software developers. | 18 |
| 11 | Simplified architecture of the application. | 29 |
| 12 | Docker application containers. | 33 |
| 13 | Multistage GitLab CI pipeline of developed application. | 37 |
| 14 | General security analysis score by Mozilla Observatory. | 60 |
| 15 | In-depth security analysis by Mozilla Observatory. | 60 |
| 16 | Skipfish results. | 61 |
| 17 | Apache killer attack simulation. | 62 |
| 18 | HTTP header with API key for reCAPTCHA. | 64 |

List of Tables

| | | |
|---|--|----|
| 1 | Percentage increase of cyber security incidents in 2017 in comparison to 2016. | 9 |
| 2 | Costs of hacking activity. | 10 |
| 3 | Layers with its accordingly protocols in the TCP/IP model. | 11 |
| 4 | Ten Most Critical Security Risks in Web Applications in 2013 and 2017, by OWASP. | 11 |
| 5 | Difference between DAST and SAST [50]. | 39 |
| 6 | Steps to load page using CSR and SSR | 53 |
| 7 | Web scanners results for developed application. | 59 |

Listings

| | | |
|----|--|----|
| 1 | TypeScript's compiler custom stricter rules. | 34 |
| 2 | Important rules for Angular's compiler. | 36 |
| 3 | Input validators in Angular. | 44 |
| 4 | Cloud Storage for Firebase security rules for file upload. | 44 |
| 5 | Cloud Storage for Firebase file upload security rules. | 44 |
| 6 | Declaration of security headers in NestJS. | 45 |
| 7 | CSP for developed application. | 47 |
| 8 | Bad code in TypeScript. | 50 |
| 9 | Clean code in TypeScript. | 50 |
| 10 | Nikto logs. | 57 |

Chapter 1

1. Introduction

This chapter introduces the context of the research. The used methodology for the practical side, scope with its limitations for the research as well as structure of this document are discussed.

1.1. Background

Since the time Tim Berners-Lee created the World Wide Web (WWW) in 1989 [1] and wrote the first browser a lot has changed. The Web environment evolved. Currently, web resources are still in a form of Uniform Resource Locators (URLs). However, Cascading Style Sheets (CSS), Hypertext Markup Language (HTML) and JavaScript on the front-end side of the browser changed how WWW looks and works nowadays. JavaScript currently is also used on the back-end side, i.e. Node.js. The client-side has evolved over the years as well, latest popularity of SPA shows it clearly. Many business solutions are served as a web platform for the client. These include products of small companies, but also enterprise-scale solutions. Thus, increased complexity of Information Technology (IT) projects based on WWW enabled dynamic development of browser environments. In those environments, SPA-based applications are becoming a common solution for the front-end layer. This is due to their Service-Oriented Architecture (SOA), modularity, performance and simplicity. They have an important role in the modern web development and as such, the Web environment itself.

The core aim of this research is to develop a secure SPA application based on Angular (8.2.0) with programming best practices. These practices are part of software craftsmanship — an approach to build high quality software. Security analysis how resistant the software is against web-based attacks by applying offensive technologies to it is a second aim.

This thesis is on the edge of software development and cyber security disciplines. Nowadays, there is a rising number of black hat activities. These areas are getting merged together in a natural way to achieve high quality source code resistant on most of the security risks. New security-based terms of software development are evolving. These are Security, Development and Operation (SecDevOps), in case of Development and Operation (DevOps)¹. Secure Software Development Life Cycle (SSDLC), in case of Software Development Life Cycle (SDLC). It clearly show that there is a need on the market and slowly willingness to pay to obtain higher security as well.

The presented degree project consists of four parts: theoretical introduction, practical software development, practical penetration testing and the security analysis for the web application.

1.2. Problem

The traditional web application is a multi-page applications (MPA), SPA takes different approach. By preloading and re-rendering all websites' elements during initialization, only

¹Software development practices to automate stages of SDLC.

displayed content, i.e. User Interface (UI), is changed during state changes. This works without the need of making additional network requests and full page loads [2]. This is an advantage over the traditional model, because user does not have to wait for the whole HTML file to be retrieved from the server. This results in higher performance (although the first load will take longer), which improves the perceived User Experience (UX). But what it means for overall security of web applications is not that straightforward.

Theoretically, a limitation of the number of Hypertext Transfer Protocol (HTTP) requests should have a positive effect on the security, at least from the network side. However, from the source code side security of a web application is dependent on using programming best practices and security testing. In-depth security analysis usually is not performed due to financial reasons. Hence, SPA-based applications are often deployed without performing these security tests. This can be risky especially in the unsafe JavaScript technology.

1.3. Goal

As part of this research a high quality SPA based on Angular framework is developed, with primary emphasis on security. The goal is to use SDLC best practices with a security first approach. Due to this, the SSDLC methodology is applied as much as possible, with in-depth analysis of the produced source code. Finally, it is measured security level using offensive technologies. The results of this project might be interesting for software developers working with Angular, JavaScript, SPA, TypeScript as well as for security architects and ethical hackers. Others technical specialists interested how to achieve security during the SSDLC and how to measure it might be interested as well.

Research question is defined as well as several sub-questions which will be addressed in this thesis:

1. How can an Angular SPA can be developed in a secure manner?
 - 1.1. What kinds of methods can be used to measure security in Angular?
 - 1.2. How to integrate software security in the SDLC of an Angular application?
 - 1.3. Do programming best practices provide higher security of web application?
 - 1.4. What are the typical security risks for web applications and how they can be mitigated?

1.4. Methodology

In order to be able to answer the stated research questions a sample SPA project is developed with features such as accessibility, internationalization and many other features relying on external dependencies. This application looks like a website of an IT consulting company which was an excellent use case in order to focus on Angular security. The reason for that is these kind of websites are complex applications with interactive and interesting elements to attract potential new customers, which results in having many external dependencies. It can be a potential source of security research for front-end layer.

The actual work starts from Angular Command-Line Interface (CLI) that has been used to set up project with extensions during development to create new classes and logic. Next,

components following the programming best practices and Don't Repeat Yourself (DRY) principle has been followed. The Behavior-Driven Development (BDD) methodology is used for development and End-to-End (E2E) testing is applied. Next, black-hat and white-hat security tests have been performed. As a first explicit security check, scanner for dependencies with known vulnerabilities has been used — npm (*npm audit*). Another tool was used to perform static code analysis. Once locally all quality elements have been achieved, automated deployment to the hosting through Hypertext Transfer Protocol Secure (HTTPS) is performed. Git is used as a Version Control System (VCS) and GitLab for repository hosting, with its Continuous Integration (CI) and Continuous Delivery (CD) capabilities. Offensive techniques to measure security once the application has been deployed include mostly penetration testing based on The Open Web Application Security Project (OWASP) top 10 security risks lists [3, 4]. The code quality checks and security tests has been automated as much as possible.

1.5. Scope

This research is limited mainly to the front-end layer, the Angular framework. Other SPA frameworks such as React and Vue.js with focus on their security should follow the same rules. From ethical hacking point of view, mainly security risks based on OWASP reports have been researched. There might be some niche or fresh types of attacks which this research does not cover. Both from development side and offensive hacking, it has been tested on main modern browsers with its latest versions, i.e. Google Chrome 76, Microsoft Edge 18, Mozilla Firefox 68, Opera 62 and Safari 12.1. Therefore, it might show different results for older and less popular browsers available on the market.

1.6. Thesis Structure

Chapter 1 provides general introduction, motivation and scope of this Master of Science in Technology Thesis. Chapter 2 is a theoretical introduction to software architecture, security and contains key concepts related to SPA. Chapter 3 explains what SPA is, with focus on Angular and explains the basic idea behind it, with metrics that measure security. Chapter 4 is the first practical chapter. General architecture of the application as well as security of Angular components are described. Chapter 5 mainly focuses on methods and tools used in order to achieve source code with high quality and security enhancements. Chapter 6 describes offensive technologies used in this research which is equivalent to practical research how secure the developed SPA is. Chapter 7, as the latest one — concludes the research and this thesis.

Chapter 2

2. Architecture and Security

For a long time software architecture and security were not combined together in terms of source code requirements. However, cyber security threats forced on software development industry to set up unified standards for architecture. It sets security as an important factor within the entire system. Architects, developers and researchers started to look for software techniques which can mitigate security risk. The reason for that is a good design assumption with historical analysis of cybercrime which can identify possible attacks. New threat scenarios based on this data could also be possible. Ignoring this fact could end up with inoperative system [5]. That is why nowadays security is an important element of software architecture already in the design stage. This is what mainly is tried to be achieved within this research, with a SPA based on Angular.

2.1. Software Architecture

Well-designed architecture helps to improve the quality of the software, by giving a clear view of how application components communicate with each other. By that it simplifies clean code techniques [6]. Combined with adopting an Agile² approach can be helpful to provide high quality software for the end user. For those purposes and many others factors software engineers community needed to invent architectural styles to simplify the work environment. Sample architectures includes: client-server, multitier, SOA, Model-View-Controller (MVC), Model-View-ViewModel (MVVM) or component-based, which all can be found in modern web applications.

2.1.1. Client-Server

An absolute basic concept of web applications is the client-server model, in which clients request certain resources from the server. After obtaining this request, servers are responsible for handling it and providing a result to the clients. This is presented by Figure 1. There are many different types of servers, e.g. file servers, network servers, print servers or web servers. Clients could be desktop/mobile applications or web browsers. In general, the WWW works with HTTP requests of which GET and POST are most frequently used for communication in RESTful Web Services (RWS)³. The browser is able to obtain files such as *.css*, *.html*, *.js* etc from web server. A GET request is normally used to retrieve information from the server, whilst a POST request is used to send data to be processed. It is noteworthy to mention that apart these two types of HTTP requests there exist also different ones, e.g. DELETE, HEAD, PATCH and PUT. They are used for applications categorized as Create, Read, Update and Delete (CRUD). Multimedia such as images and videos are served as a response from the web server includes another use case. One more could be submitting a form of data and waiting for specific response.

²A method of project management that simplifies meeting requirements of a clients to deploy highly valuable software from technical and business aspect. It is done by analysing and improving developed product already on stage of development.

³Web services conforming with REpresentational State Transfer (REST), REST — architecture style for managing state information

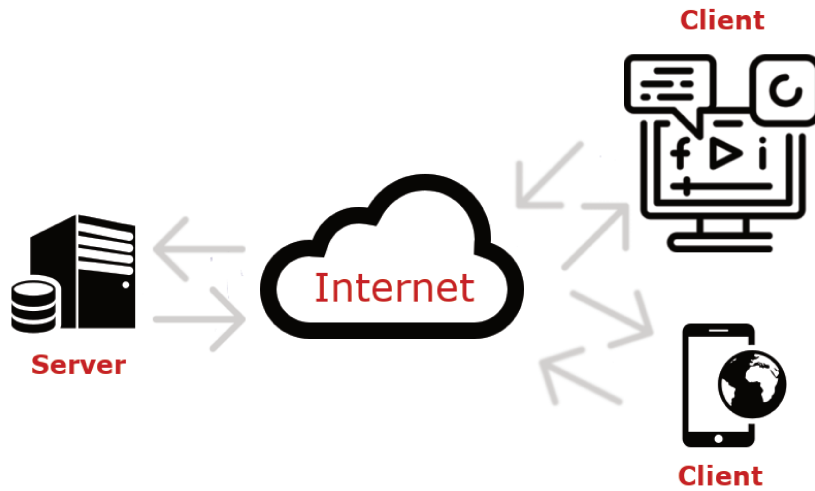


Figure 1: Client-server architecture in the Internet.

2.1.2. Multitier architecture

Requesting resources for every asset sequentially and retrieving responses from the server usually is not enough in the current environment. Complex cloud solutions, modern mobile solutions and web applications contain a lot of logic. They should be optimized to improve maintainability and performance as well as to simplify development. In order to achieve scalable applications modular programming was introduced, in which each logical unit of the application is separated into individual modules. This is a foundation of multitier architecture. In that particular architecture application, data management and presentation tier are separated into different tiers. Such architecture with different tiers is often present in applications are developed using modern architecture guidelines. An example of such logical division is presented by Figure 2. The top tier is a presentation tier, it is responsible for showing the actual content for a user. Application tier contains business logic including authentication, security and other middleware. Persistent tier is a presentation tier for data. The bottom tier is optional and holds for example databases.



Figure 2: Multitier architecture, often present in modern applications.

2.1.3. Service-Oriented Architecture

Increasing technical needs from software development industry for more complex projects required more sophisticated architectures. SOA helps to improve efficiency, scalability and management for large software projects [7]. The idea behind SOA is to divide logical parts into smaller chunks called components. Important fact is that each of them should be independent and limited as much as possible. In such a way changes in one of them should not influence other components. SOA focuses on defining strategic goals, flexibility, providing business values and shared services with its inter-process communications between each other [8]. Typical implementations of SOA often are performed using web services, e.g. REST and Simple Object Access Protocol (SOAP). This architecture style finds industrial application especially within enterprise-level applications, where the scope of the project is quite extensive. It helps the architecture to be more agile-oriented, cost-effective and flexible.

2.1.4. Model-View-Controller

Architectural pattern designed for desktop applications, later adopted in many others environment such as the web and mobile applications is MVC. It has three core elements: Model, View and Controller. The first of them — Model, contains business logic, handles application state, reads and writes data as well as validates data on the front-end layer. The View, as its name suggests, presents certain views to the user based on data received from a controller. Its tasks includes the interaction with the user. The connection between model and views is provided by the Controller. This is a kind of a bridge between those two, as can be deduced from Figure 3. The Controller can be understood as a managerial part of the application, as it interacts directly with both of the other elements of this architectural

styles. The last one plays an important role, because the View and the Model should not know about each other; indirect communication between View and Model should be done through the Controller.

An obvious advantage of this architectural style is separation of content into different logical elements. Similar idea exists with the previously described multitier architecture. It helps to organize source code into different layers, by that cleaner code is easier to achieve. Maintenance of such application with its future extensions is also improved. Easier modifications, logical grouping, low coupling⁴, simultaneous development or reusability are definitely positive sides of MVC. The view should not handle the business logic, because it is responsible just for showing the actual content. The model should focus solely on business logic, adding such methods to the controller will cause it to be overwhelmed after some time. The controller is a binder between these two.

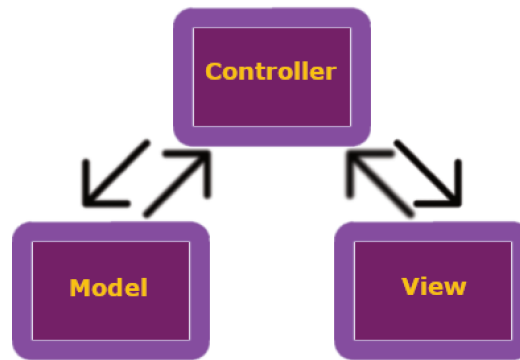


Figure 3: MVC architectural style.

2.1.5. Model-View-ViewModel

The MVVM architecture is an important concept in terms of this research, since Angular itself follows many of the MVVM rules. The ViewModel main tasks include managing the model and transferring its data to the view. Simply speaking it is responsible for transforming the data to such format the view can present to the user in an acceptable structure. The view elements are also different. In MVC it is more passive, whilst in MVVM it contains events and data binding⁵. Views are not maintaining their states. For this task the ViewModel element is responsible to synchronize it with the view. Sometimes MVVM is also known as Model-View-Binder, due to the reason that it contains a binder. It controls communication between the view and the ViewModel. The binder also helps the developer to follow certain rules. Only exposed properties can be accessed by the ViewModel. Thus, it disallows breaking a good practices in an architectural style. Figure 4 shows MVVM architecture with its basic idea.

⁴Writing software in such a way that all classes work independently without relying on each other.

⁵Establishing connection between the user's view and business logic implemented in a model.

Arguments for MVVM include better separation of the view layer from rest of the application. It makes easier for people working on the presentation layer, i.e. UI and UX designers. Instead of dealing with the business logic they can focus on the user interface itself. This way, it is more convenient to work on multiple tasks simultaneously, because each element of the MVVM architectural style has clear separation of functionalities. Last, but not least a positive argument for introducing such separation is an improved testability of such application. This is especially helpful for teams working with BDD and Test-Driven Development (TDD) approach.

Criticism against MVVM includes unnecessary overcomplication for smaller projects, where the UI layer itself is not too complex. Apart from that, another argument is that in larger applications there exists a problem with managing those layers once the source code grows to an enormous size. Moreover, such big projects can have difficulties with substantial memory consumption, due to many data bindings in the application. Due to this reason a component-based architecture has been implemented in the Angular framework.

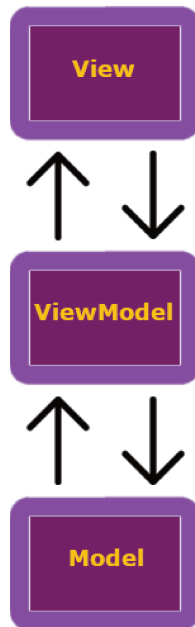


Figure 4: MVVM architectural pattern.

2.1.6. Component-Based Architecture

The architectural model implemented in Angular is a component-based architecture. It has some similarities with others architectures described earlier. Components are a modular,

portable and reusable set of functionalities. This is an encapsulated entity of specific behaviours, styles and usually with one view to represent those.

The main focus is on component reusability, as it is shown on Figure 5. Each logical part of an application is a separated component. However, defining one component should not collide with reusing it. In the following example a *Call To Action (CTA) component* has been used twice. With the component-based architecture it is defined only once without code duplication. The application is instructed simply to show the same component twice, but in two different parts of the application. These reusable components reduces codebase, follows DRY principle and, ensures Separation of Concerns (SoC).

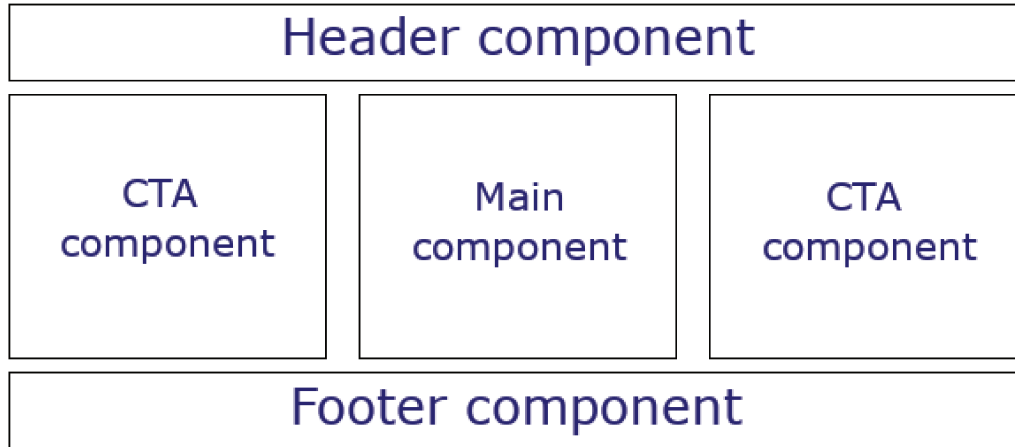


Figure 5: Component-based architectural pattern.

2.2. Software Security

Cyber security is a relatively fresh part of IT. One of important parts in it is software security. This discipline is becoming more and more important. This is due to many reasons, i.e. big data leaks from companies such as Facebook [9], Marriott [10], Uber [11], recent cryptocurrencies popularity [12], increasing cybercrime activity as well as introduction of General Data Protection Regulation (GDPR) [13] by European Union (EU). Tables 1 and 2 show key statistics for cyber security market based on year 2017 [14].

| Name of a security-related term | Percentage [%] |
|---|----------------|
| Increase of attacks on Internet of Things (IoT) devices | 600 |
| Annually growth of ransomware | 350 |
| Companies keeping old sensitive files | 74 |
| Users were never asked to change passwords | 65 |
| China, as a source of attacks | 20 |
| United States, as destination of attacks | 18 |

Table 1: Percentage increase of cyber security incidents in 2017 in comparison to 2016.

| Name of a security-related term | Financial consequences [\$] |
|--|-----------------------------|
| Foreseen damage to hacking-related activities in 2021 | 6 trillions |
| Average cost of cybercrime for financial services industry company | 18.3 millions |
| Median cost of malware attack on a company | 2.4 millions |

Table 2: Costs of hacking activity.

Published in 2006, the book by Gary R. McGraw [15] was an eye opener to develop software in a secure manner. The author defines seven security touchpoints of secure software which is presented on Figure 6. These points include: abuse cases, code review (tools), penetration testing, risk analysis, risk-based security tests, security operations and security requirements. Although the book was published several years ago, from an architecture point of view the content remains valid, just the described tools may have changed. Hence, the ideas presented in the book will be the foundation for security requirements in this research project.

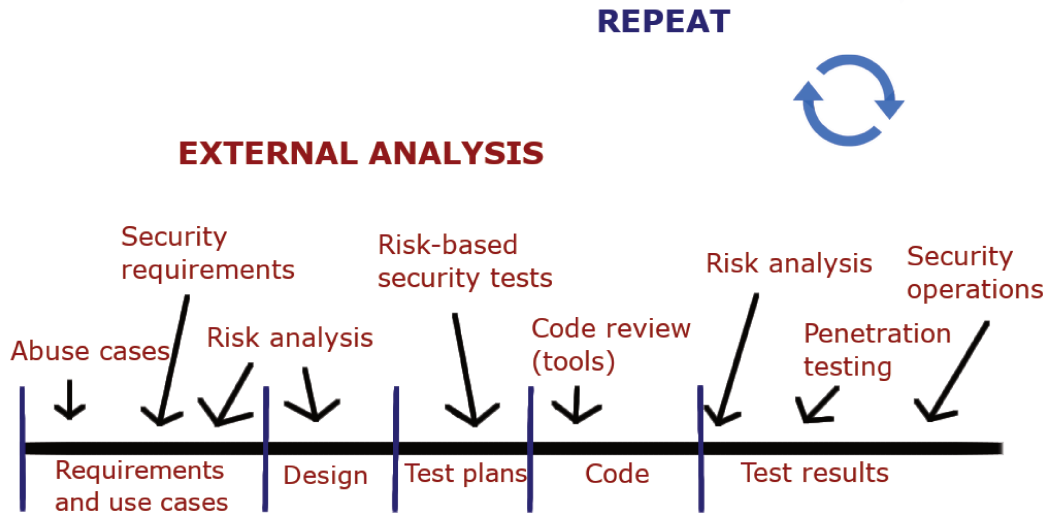


Figure 6: Seven software security touchpoints [15].

The security touchpoints are used depending by stage of the project, i.e. architecture and design, code, feedback from the field, requirements and use cases, test plans, tests and test results. Depending on the level of advancement different touchpoint should be used. For example, for test results firstly risk analysis should be used, then penetration testing and finally security operations. When the software is already ready to be deployed — in the final stage, it is more difficult to increase security. An extra touchpoint with added value could be analysis by an external team of security specialists, who have not contributed to the code. Often people not engaged in the project are able to find mistakes easier. The

whole process should be repeatable, it can be seen as a beginning of software craftsmanship.

2.2.1. Risks

Security of software is dependent on many factors, especially in the web where many logical layers exists. A structure of these layers can be represented in terms of the Internet Protocol Suite (TCP/IP), as Figure 3 shows. Different layers are managed by different protocols. This figure shows only a limited number of protocols, it is not an exhaustive list. Numbering layers in TCP/IP model starts from the bottom. Link layer in TCP is the bottom one, while application layer is the top layer etc.

| Position | TCP/IP layer name | Typical protocols |
|----------|-------------------|-----------------------------|
| 4. | Application | DNS, FTP, HTTP, HTTPS, IMAP |
| 3. | Transport | TCP, UDP |
| 2. | Internet | ICMP, IP |
| 1. | Link | ARP, Ethernet, MAC |

Table 3: Layers with its accordingly protocols in the TCP/IP model.

Due to the reason the WWW has a very complicated structure, some kind of report to make developers aware about security was needed. Currently OWASP provides insights of security for the web. Based on their cooperation with community, industry and researchers they are publishing every few years list of top ten security risks. Recent lists are from 2013 [3] and 2017 [4], presented by Figure 4. The order of the risks is according to their popularity, e.g. injection is the most popular type of security risk for both reports.

| Position | OWASP Top 10 — 2013 | OWASP Top 10 — 2017 |
|----------|--|---|
| 1. | Injection | Injection |
| 2. | Broken Authentication and Session Management | Broken Authentication |
| 3. | Cross-Site Scripting (XSS) | Sensitive Data Exposure |
| 4. | Insecure Direct Object References | XML External Entities (XXE) |
| 5. | Security Misconfiguration | Broken Access Control |
| 6. | Sensitive Data Exposure | Security Misconfiguration |
| 7. | Missing Function Level Access Control | XSS |
| 8. | Cross-Site Request Forgery (CSRF) | Insecure Deserialization |
| 9. | Using Components with Known Vulnerabilities | Using Components with Known Vulnerabilities |
| 10. | Unvalidated Redirects and Forwards | Insufficient Logging & Monitoring |

Table 4: Ten Most Critical Security Risks in Web Applications in 2013 and 2017, by OWASP.

One of the crucial elements in web security is associated with transferring data between web services. Web services are nowadays mostly based on the RESTful architecture style. This architecture style is simple and flexible way to design Application Programming Interfaces (APIs), called RESTful APIs. REST takes advantage of existing protocols (HTTP). Response objects are usually in JavaScript Object Notation (JSON), eXtensible Markup Language (XML), YAML Ain't Markup Language (YAML) or other text-based formats. This is an advantage over SOAP which defines responses to be returned with a XML format. Even though SOAP is an official web standard, REST is getting more popular. Its

complexity is smaller, it provides higher performance and scales easier. SOAP is a protocol, while REST is an architecture style. However, they both can be used for the same purpose and thus can be compared with each other. An example of the SOAP is the PayPal SOAP API [16], while for the REST it is WordPress REST API [17]. The reason for REST's popularity might be its simplicity in comparison to SOAP, but it comes with the price of security. SOAP comes with standardized security rules such as built-in Atomicity, Consistency, Isolation, Durability (ACID) compliance or authorization provides higher level of security using when implementing its guidelines called *WS-Security*. Due to them SOAP enables security implementation in more standardized way. REST supports JSON parsing which is usually faster than parsing XML [18]. This is one of the most valuable advantages over SOAP and could be one of the reasons why it is more popular than SOAP. Especially given the popularity of mobile devices, as for those end user every second matters. However, in enterprise-level web services which require higher security, e.g. financial services or in telecommunication companies, SOAP is still the preferred way for data communication.

The progressive popularity of mobile and web applications, mostly based on REST, enforced OWASP to prepare special requirements for software which uses REST as a method to communicate between web services. The good security practices recommended by OWASP [19] include:

- Access control — access control has to be enforced in all micro services which is a bit more complicated in comparison to monolithic applications.
- API keys — to protect endpoints API keys should be used.
- Audit logs — analysis of token validation errors with security alerts are helpful to detect attacks.
- Cross-Origin Resource Sharing (CORS) — a mechanism which allows website at specific domain to have permissions to access some resources from a different domain.
- Error handling — error message can be a source of potential helpful information for an attacker, thus revealing technical details about them is a security mistake.
- HTTP return codes — semantic status codes should be used for responses in order to follow proper HTTP specification for different scenarios.
- HTTPS — each API endpoint must provide HTTPS to protect authentication data such as API keys or user credentials.
- Input validation — by default untrusting input parameters is a good practice, thus entered data should be validated. Only strictly allowed can proceed further, suspicious input must be rejected.
- JSON Web Tokens (JWT) — protection using digital signatures⁶ or by Message Authentication Code (MAC) is required to protect this standard, because they are often used for security tokens.

⁶Called also cryptographical signatures, those are mathematical methods to verify authenticity of a certain message.

- Management endpoints — strong protection must be delivered by Access Control List (ACL) or firewall rules. When accessing online multi-factor they should be hided from the Internet.
- Restrict HTTP methods — rejecting all HTTP methods that are not on the white list with a response code 405, i.e. *Method Not Allowed*.
- Security headers — returning *Content-Type* header with *charset* causes correct interpretation of the content by the browser and the server's correct answer also is important against XSS.
- Sensitive information in HTTP requests — to avoid leakage of secret API keys, credentials or security tokens cannot be shown in the URL. Instead, GET, POST and PUT methods should handle it in their appropriate sections to keep sensitive data.
- Validate content types — validating incoming requests and sending safe responses with appropriate content types are required for correct interpretation on the client-side.

There are many security risks on different layers which can cause to defeat security defenses. Thus, in the software industry information about many threats can be heard. These mostly has been caused by exploiting vulnerabilities. Often, even one weakness in the application was enough to break it totally or cause a data breach. As the list shows, there are many parts to take care of.

2.2.2. Vulnerabilities

General issues related to security are based on *vulnerabilities* — possible weaknesses in a software which can be exploited by an attacker [20]. They should be distinguished from security risks, those two are different terms. Currently many applications are based on cloud solutions, available through different types of services. These includes Infrastructure as a Service (IaaS), Platform as a Service (PaaS) or Software as a Service (SaaS). Vulnerabilities can occur on many different layers. Beginning from web application core source code, through network-related problem, all way to cryptographical methods. Such complicated combination of layers in a typical web-based project leads to many possible weaknesses. They are caused by lack of testing, obsolete encryption algorithms, poor storage management, programming mistakes, unencrypted communication protocols traffic, usage of APIs with malicious code, weak passwords, wrong databases configuration [21]. Monitoring of Common Vulnerabilities and Exposures (CVE) is a valuable source of potential security risks. Lists of those are published by National Institute of Standards and Technology (NIST) [22]. The records are based on specially designed for it website [23] which is kept up-to-date by the community and security researchers. From developers perspective it is important not to use dependencies and software versions which contains any kind of weaknesses. Automatic monitoring as well as updating them as fast as possible once security updates are deployed is a typical countermeasure.

2.2.3. Threats

A danger with possible harmful to certain system complications due to intentional or accidental activity is called a *threat* [24]. Effects of a successful hacking activity can compromise availability, confidentiality or integrity. Typical threats are botnets, computer worms, fake

security software, malicious spyware, malware, phishing, rootkits, spam, Trojan horses and viruses. Most of them target machines, but some of them humans. Machines usually respond automatically, thus their security relies on the software which is supposed to defend from malicious activity. Humans should follow security policies, have basic knowledge or at least installed antivirus software with up-to-date virus definitions database.

2.2.4. Control

Assurance of security requires that developers follow certain rules and should be controlled through earlier established standards. Comprehensive list of preventive measures has been published by OWASP in 2018 [25] and includes:

1. Define security requirements.
2. Leverage security frameworks and libraries.
3. Secure database access.
4. Encode and escape data.
5. Validate all inputs.
6. Implement digital identity.
7. Enforce access controls.
8. Protect data everywhere.
9. Implement security logging and monitoring.
10. Handle all errors and exceptions.

These rules are easy to follow not only on the stage of development, but also during validation, i.e. when assessing how secure the application is. Web applications contains many layers of source code present in business logic, controllers, databases, views etc. That is why ensuring high level on security includes to meet the requirements on all those layers. Modern security defenses have many layers of security. It means that breaking into one layer does not mean the whole system is broken. Keeping security in mind is easier with these guidelines, from security analysis point of view as well as software development.

2.2.5. Secure Software Development Life Cycle

High quality source code with agile-oriented way of working is called SDLC. This is presented on Figure 7. Its first stage starts with planning requirements for the project, followed by defining requirements, designing the product architecture, developing, testing and deploying the product. It can be easily implemented into Agile methodology, which will render the SDLC to become iterative.

On top of the framework designed for security requirements a new model has been proposed called SSDLC. Its basic concept is presented by Figure 8. Difference between those two models by naming includes the first two stages, where in SSDLC security is mentioned with a priority. The security definition phase would include planning as well. The research phase would define general concept of that application. However, even though the next

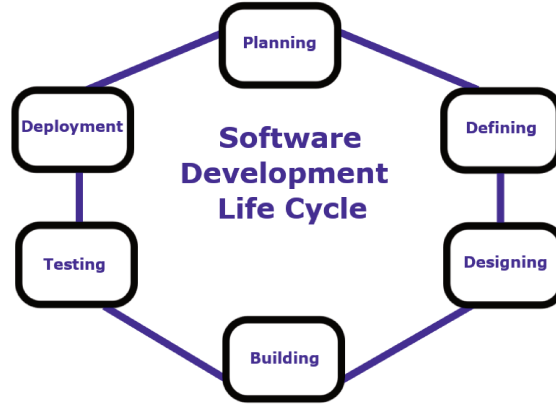


Figure 7: SDLC methodology.

phases look exactly the same, their tasks will be slightly different. With more focus on security, e.g. for testing apart functional tests, fuzzing⁷ or mutation testing⁸ could be added. Similarly to the SDLC, its extension called SSDLC can be smoothly implemented into teams working in Agile, making it an iterative process.

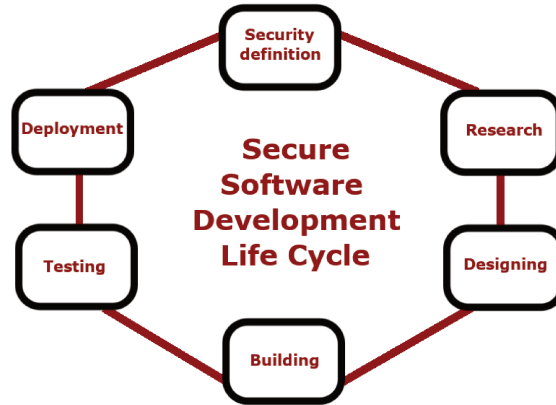


Figure 8: SSDLC methodology.

2.2.6. SecDevOps

Prioritizing security on the first position in DevOps projects is getting called SecDevOps. Apart this, there are DevSecOps (Development, Security and Operations) and DevOpsSec

⁷Providing invalid, random and unexpected inputs.

⁸Changing (mutating) statements in a source code and observe if there are any errors.

(Development, Operations and Security).

The main difference between these three terms, i.e. SecDevOps, DevSecOps and DevOpsSec is the *Sec* part. In terms of security this is crucial, because it defines on which stage of the SDLC it should be placed. For SSDLC the correct way supposed to be only SecDevOps. In terms of the SecDevOps security is defined already on the earliest stage of a project. The DevSecOps predicts security checks after development. DevOpsSec will take security into considerations once the application will be deployed.

Improving security in the last part of product development might be difficult to achieve. The reason for that is when all the programming is already done and deployed as designed, changes for security reasons would need to imply changes on the architecture level. Enterprise-level applications might be very difficult to change on that level, sometimes it is even impossible to fully integrate security, as it should be. In a positive scenario it would take significant amount of time to make it properly. However, it is supposed to be the easiest solution. The currently popular DevOps technique is present in many companies delivering modern applications. In these projects which have started before SecDevOps was born it is kind of natural way to extend it like so and thus follow DevOpsSec.

DevSecOps makes security a higher priority than DevOpsSec. In that model it is placed directly after implementation, before deployment. In terms of security it is a more correct approach, because the application is not delivered to the commercial environment before passing necessary security tests. This approach causes similar problems. Once security flaws will be discovered, then sometimes it might be quite difficult to increase security without changing the architecture significantly. However, in terms of security this is much better approach.

Including code reviews, proper cryptography usage, defensive techniques based on OWASP guidelines, programming best practices, secure access control, static code analysis, threat modeling, vulnerability assessment and others security-related mechanisms at the starting stage of an IT project is what SecDevOps is about. In this technique security requirements are defined before development starts. The problem of completely redesigning the architecture due to security changes does not exist, because it is defined during first stage of the project. Only new vulnerabilities have to be taken into consideration, but with security first approach it is easier to manage source code, than vice versa. It might come with a cost of slower development. However, in a long term period decreasing potential of security issues results in a smaller amount of successful attacks. They become more and more expensive once getting older.

Chapter 3

3. Single-Page Applications

The traditional way to build web applications is based on a MPA approach. In the MPA for any action page is loaded. A different approach is taken by SPA, where the client loads all the resources during first load only. Therefore, it does not need page reloads during events such as clicks, routing, sending forms etc. Even if the browser is closed, then due to cache mechanism it loads fast for the second time and later on. By that it provides better UX for the client-side. Currently many modern web applications follow the SPA model.

3.1. Web Applications

Typical layer separation for web applications is done by separating the front-end layer — presented in a browser, the back-end layer — handled by a web server and the database layer. Figure 9 clearly shows relationship between those three. The browser is responsible for displaying the content to the user. The server handles requests from the front-end layer and sends responses back to the browser. The database layer stores data which may include user credentials hashed values. SPAs are naturally also considered to be web applications. They are placed on the browser side, however due to their specification they are able to provide a rich front-end experience. In the JavaScript world typical technological stacks on the front-end side includes Angular, React or Vue.js. The server layer can be implemented by Node.js with Express. MongoDB is an example of natural database for the JavaScript stack. However, the back-end could be also any different technology, e.g. C#, Go, Java, Ruby on Rails etc. The JavaScript-related technologies plays nicely together, but any other back-end and database could be used as well.

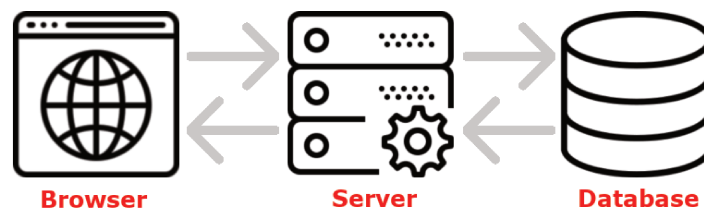


Figure 9: Logical layers separation of a typical web application.

3.1.1. JavaScript

In 1995 the programming language JavaScript originally was introduced to improve interaction on the client-side of web application in a browser. Over many years it changed significantly. Currently is the most frequently used language in the world, as can be seen on Figure 10. Not only the front-end layer is currently a domain of JavaScript, but it can also be found on the server-side (Node.js). Building full stack web applications can be done using MongoDB, Express, Angular and Node.js (MEAN) stack. Other use cases includes:

cross-platform solutions (Electron, Meteor), mobile applications (Apache Cordova, Ionic), Progressive Web Applications⁹ (PWA) or even IoT (IoT.js).

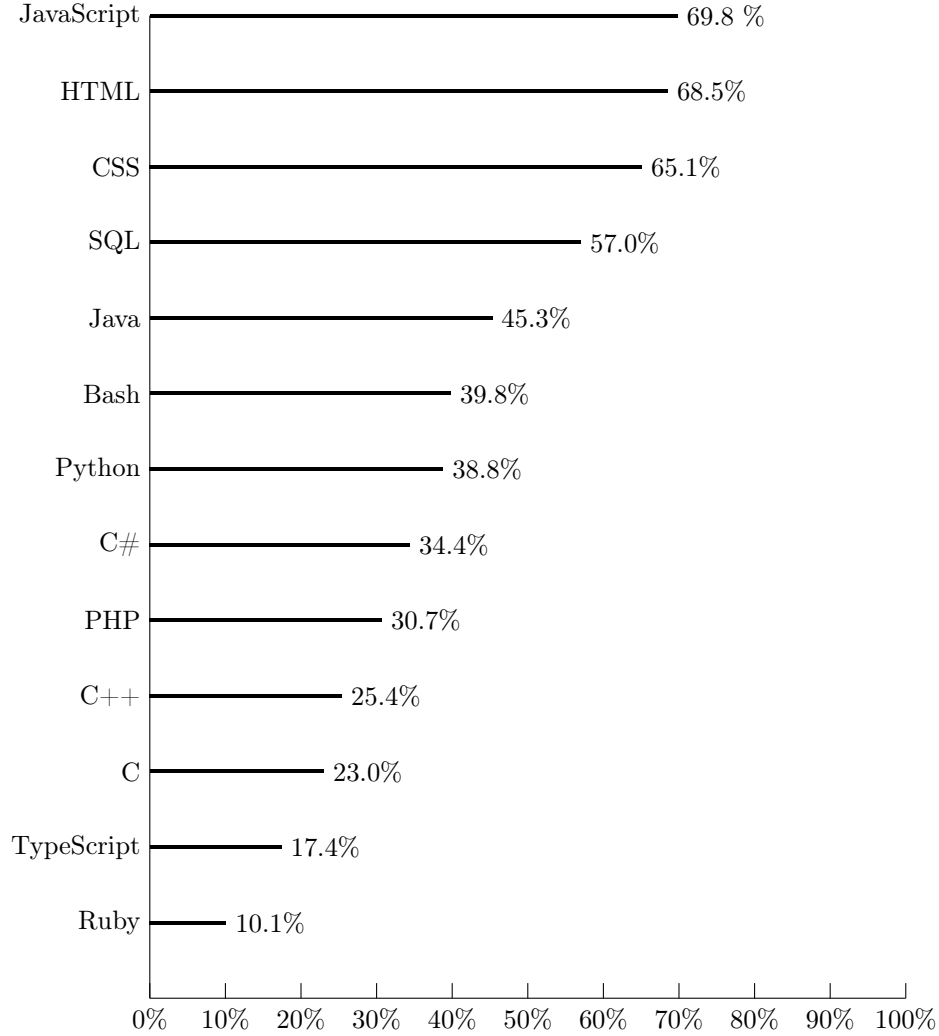


Figure 10: Most popular technologies, annual report by Stack Overflow for 2018 [26]. Includes languages with at least 10% usage by software developers.

JavaScript is a high-level, interpreted language which complies with ECMAScript — a scripting-language specification on which JavaScript is based. Basic features of this language include: delegation, dynamically typed, functional, object-based, supports structured programming and wide usage of prototypes. For a long time it has been considered as a non-secure programming language due to its unsafe nature. However, its recent popularity also led to interest in the topic of security in this certain technology. Combination of Document Object Model (DOM) with its manipulation by JavaScript are potential source

⁹Web applications with access to mobile native applications functionalities.

of vulnerabilities. Possible attacks are CSRF, XSS or other client-based possible exploits. However, this is not limited only to the front-end layer of web applications. As mentioned previously, JavaScript has been applied in different environments. Hence, its security becomes more and more important. Especially when handling authentication, authorization, sensitive data or payments in the business logic layer.

Its ecosystem changed significantly over past several years. Currently plain JavaScript (without any frameworks or libraries, also known as *Vanilla JS*) is not often used in developing modern applications. Frameworks such as Angular, React and Vue.js or libraries like jQuery, RxJS and Underscore.js are daily used by hundreds of thousands of developers. Moreover, workflow for this specific language includes its specific tools, such as Babel, Bower, ESLint, Grunt, Gulp, Karma, npm, webpack, Yarn, Yeoman etc. They automate certain tasks, bundle source code for deployment, check syntax, transpile into a syntax readable by the browser and takes care of many other processes.

3.1.2. TypeScript

Angular is built in TypeScript, a programming language introduced by Microsoft in 2012. TypeScript is recognized and advertised as a superset of JavaScript. As its name suggests — it is strongly typed language. Apart strong typing other features includes asynchronous pattern, classes, enumerations, generics, interfaces, namespaces, modules, tuples, typed annotations and typed inferences. Angular applications projects source code is written in TypeScript and transcompiled to JavaScript. Such kind of technique provides a way to execute written code by a browser. It will be the main programming language used to develop the practical side of this research project. However, once the application is deployed JavaScript is the language present in the browser responsible for executing it properly. TypeScript is not natively supported by the browsers, at least at this time of writing this thesis.

3.2. Frameworks

Often modern web applications are taking advantage of using frameworks, external libraries or both. Frameworks defines the structure of an application. Libraries provide ready functions to be used by the developer, without the need to work in a plain language, e.g. Vanilla JS. Nowadays, developers communities mostly use frameworks such as Angular, React or Vue.js on the front-end. Frameworks are important additions to programming languages because they motivate on developers to follow certain rules. Using them, projects are easier to follow and maintenance becomes simpler as such application have a more structured architecture. Efficient and scalable SPAs are usually based on such frameworks. In this thesis, Angular is studied. Its environment is mature, provides many best practices, good technical documentations, as well many problems (including security) has been solved since that time Google introduced it in 2016. The Angular framework should not be mistaken for AngularJS framework, which was introduced by Google in 2010 but in fact is a completely framework with a completely different approach solving different problems.

3.3. Angular

One of the most popular frameworks for developing SPAs is currently Angular. It is not only possible to use Angular for web applications, but with its cross platform approach Angular

can also be used to create desktop and mobile applications. AngularJS follows the MVC architectural pattern, but Angular does not. Moreover, it also does not follow MVVM, but takes many practices from it. Instead, Angular follows a component-based architecture. Angular consists the component itself, its View and business logic. Such components must be added to declarations in a parent module, then assigned to the View, i.e. *.html* file and is rendered by the browser.

Angular provides many built-in features, e.g. animations, code optimization techniques like Ahead-of-Time¹⁰ (AoT) or lazy loading¹¹, dependency injection¹², form validation, internationalization, offline capabilities, security mechanisms, simplified HTTP API and type safety (TypeScript). Thus, it is a good choice for scalable applications, even though it has its specific workflow. At the beginning this workflow can be seen as a disadvantage over other frameworks, but such kind of mature ecosystem is incredibly important when releasing stable software. There is no need to rely on third-party packages, because the core functionalities provided by Google are already forming ripe environment to create wide spectrum of web applications. It is still possible to extend working environment by using one of thousands of extensions developed by the community. It clearly shows how popular Angular is.

Newly introduced mechanisms such as Angular Ivy¹³, i18n¹⁴, tree shaking¹⁵ are definitely interesting. However, even more helpful changes for developers are coming. One example is adding to the Angular CLI default renderer based on the Angular Ivy engine. This clearly shows Angular's potential. Google keeps, apart from having an internal, dedicated team for Angular improvements this framework open source. This helps the community to contribute with their ideas to further improve it. One of successful ideas coming from the community includes introduction of NgRx, a reactive state management inspired by Redux¹⁶.

Another reason to create a SPA based on Angular is its willingness to test wide spectrum of SPA features. These could be potential source of security vulnerabilities. Hence, more features, such as accessibility or offline capabilities can be integrated using the latest version of Angular and validated how secure the application is.

3.3.1. Security

There are several best practices for security in Angular applications such as using the latest version of this framework, do not make any direct modifications to the core of Angular and avoid usage of flagged as a security risk APIs. However, security on the front-end layer is more than that. Terms such as Content Security Policy (CSP), CORS, CSRF, Cross-Site Script Inclusion (XSSI), data sanitization or XSS are one of possible vulnerabilities developers can face.

¹⁰Compilation process to produce efficient JavaScript source code during build time to speed up rendering in the browser.

¹¹Approach to defer loaded content to the moment when it is actually needed, instead of loading them upfront.

¹²Technique for producing loosely coupled code with supplying dependencies to another object.

¹³Rendering engine responsible for providing smaller bundles faster.

¹⁴Internationalization capabilities.

¹⁵Dead-code elimination.

¹⁶JavaScript library for managing application state.

XSS is an attack, in which malicious code can be injected into an application. Prevention includes defending the DOM from ability to insert attacker's code there, e.g. using a *script* tag. By default Angular classifies all values as untrusted, while inserted values via DOM are sanitized. That way, suspicious values are escaped by its framework design. Apart from possible client-side XSS attacks, there also exists server-side XSS. For these different type of protection should be ensured. For that reason generating Angular templates on the back-end layer by using templating languages can be a source of unconscious template-injection vulnerabilities. In terms of security it is better to use automated escaping of values provided by templating language itself, but not by generated Angular templates based on it.

The security model for XSS includes also using an offline template compiler to avoid template injection. This technique also improves performance. Dynamically generated templates based on concatenation of user input and templates are not considered to be secure. They bypass built-in security protection of Angular.

Data sanitization is one of methods to prevent XSS attacks. It is performed to filter untrusted values from source code. Angular recognizes six security contexts, i.e. HTML, none, resource URL, script, styles and URL. An important note is that resource URL is not sanitized by Angular, because it is simply not doable. Sometimes developers have to access DOM APIs directly which is not recommended in terms of security. Hence, in that scenario built-in sanitization methods should be used, i.e. *sanitize* of the *DomSanitizer*¹⁷ with adequate security context. This technique is crucial, because native DOM APIs by default do not provide protection against security vulnerabilities. Therefore, manipulating DOM via *Renderer2*¹⁸, instead of using *ElementRef*¹⁹ is considered to be more safe. Dependencies are key elements when developing SPAs. Third-party APIs might have calls to unsafe methods such as *ElementRef*. Developers could assume that unit testing will provide clean code and it is all in terms of security. However, those are isolated tests which do not touch external dependencies the community has to rely on. Hence, this is another reason to analyse security in Angular in more depth before deployment.

Next protection against XSS attacks is to introduce CSP into an application by permitting only allowed origins to be executed. Let consider an example, an URL *https://my-trusted-source.com* is whitelisted by a web application *https://myapp.com*, thus considered safe. That is why execution of scripts from *https://my-trusted-source.com* on *https://myapp.com* is permitted. On the other side, there exists a harmful sample script from *https://malicious-code.com*. It has been not whitelisted by the *https://myapp.com*. Therefore, *https://malicious-code.com* cannot execute its scripts on *https://myapp.com*. Defenses against XSS in terms of CSP include the previously mentioned source whitelists. It is achieved using CSP-defined set of policy directives (e.g. *default-src*, *style-src*, *script-src*, etc.) with whitelisted URLs [27].

Apart from that, good programming practices have a positive impact on security as well. The first one is to disable executing *eval()*²⁰ and minimize inline codes. It might lead to detrimental behaviour of web application. These introduces the risk of DOM-based XSS²¹

¹⁷Security helper for XSS prevention when sanitizing values to be used in different DOM contexts.

¹⁸Abstraction for manipulating DOM without directly accessing it.

¹⁹Wrapper around native DOM element.

²⁰Method for evaluation or executing an expression.

²¹Modifying DOM environment by executing scripts in victim's browser using client-side script.

vulnerability due to unsanitised values. That is another reason to use separated *.js* files. Usage of *eval()* seems to be outdated, however there are different object used in JavaScript through which text can be injected, i.e. *new Function()*²², *setInterval()*²³ or *setTimeout()*²⁴. Potentially, this could end up with injection of malicious content. Thus, CSP by default blocks Strings in these functions and calls for these three methods should be written as inline functions rather than Strings [28]. Information about security on the back-end side is important as well. By using CSP developers can set up reports through POST requests in form of JSON. Such response contains location (route) of a reported vulnerability, suspicious URL, resource of harmful script, what kind of directive it violates and page policy. Specifying CSP can be done in HTTP header from web server. Alternatively, it can be also enabled using the *<meta>*²⁵ tag attribute in HTML with *http-equiv*²⁶ attribute.

One of two most frequent HTTP vulnerabilities is CSRF. It is one of the OWASP Top 10 security risks in 2013. This type of attack combines social engineering techniques and executed harmful script on the desired by an attacker link. In such type of attack the hacker tries to trick the victim to visit a different page than the person wants, e.g. infected *https://attacker.bank.com*, instead of trusted *https://bank.com*. In such a manner an attacker via its controlled website, i.e. *https://attacker.bank.com* is able to manipulate a specific action, e.g. transferring money from victim's digital wallet to its own account. At a first glance to defend against such attacks does not seem to be difficult, however some of the defenses might cause security leaks [29], such as:

1. **Using secret cookies** — important to remember is the fact that secret cookies will be transferred as well with every request. As a result, authentication tokens will be also submitted with session identifiers. In such a way session identifiers are not helpful, because they do not verify if the end user wanted to submit the request.
2. **Only accepting POST requests** — even though a web application would accept only this kind of requests, hidden values in a form on website controlled by an attacker might not be able to resilient against CSRF.
3. **Multi-step transactions** — in a scenarios when an attacker can guess the next step, then it cannot defend the victim.
4. **URL rewriting** — even though an attacker cannot guess session ID by introducing URL rewriting of the victim it would reveal the session ID in the URL.
5. **HTTPS** — usage of HTTP over Transport Layer Security (TLS) is only a solid background for ensuring security in an application, but itself does not defend against CSRF.

There are two methods of CSRF attack resistance, i.e. token based (stateful or stateless) and user interaction based protection (one-time token or reauthentication) with token based mitigation [30]. Most frequently used and most recommended is token based mitigation. The stateful technique is achieved using synchronizer token pattern. Stateless, as an encrypted or hash-based token pattern. Those techniques require practical and well-designed

²²Built-in constructor for defining a method.

²³Method for calling certain logic at specified interval.

²⁴Method for evaluating an expression after specific number of milliseconds.

²⁵HTML tag for providing metadata about data.

²⁶HTML attribute for providing HTTP headers.

usage of cryptography. Secondly, due to those strong foundations of cryptography it is easy to make a mistake. That is why developers should use deployed implementations which have been reviewed and tested by the community of security experts. Many of those defenses have been described by an important paper over a ten years ago [31] and the basic concepts are still up-to-date.

Angular provides built-in protection against CSRF [32] via its *HttpClient*²⁷. This mechanism relies on reading tokens from cookies during HTTP requests. Scripts outside the trusted domain should not be able to read those cookies. Therefore, server-side can be assured it did not come from an attacker's domain²⁸. The default configuration of an interceptor²⁹ is set to accepting POST requests and other mutating requests with relative URLs. GET and HEAD requests as well as absolute URLs are rejected. Moreover, for that to be working correctly server of the hosted application must set an *XSRF-TOKEN*, a readable session cookie on page load or the first GET request. Based on this the server is able to ensure that the cookie is valid if it matches appropriate HTTP header, i.e. *X-XSRF-TOKEN*. It ensures that only scripts within specific domain could send this request. Preventing clients from creating their own tokens is achieved by server verification and uniqueness of the token for each user is important. Additional security might include adding salt³⁰, while forming a token to a hash code³¹ of the authentication cookie for the developed website. However, this is only the front-end side of CSRF protection. For effective default security defenses against CSRF the back-end also has to be configured properly. This can be achieved by setting cookies for the page and verifying if desired header is available in all required requests. Basically, this technique is achieved using same-origin policy. This is already doable from technical point of view, due to the reason that it has been implemented by modern browsers.

However, sometimes developers have to request a resource from a server outside their domain. A typical use case is accessing an external API and in such scenarios CORS can be used to handle those requests. It is achieved by setting up HTTP response headers such as *Access-Control-Allow-Origin*, *Access-Control-Expose-Headers*, *Access-Control-Max-Age*, *Access-Control-Allow-Credentials*, *Access-Control-Allow-Methods*, *Access-Control-Allow-Headers*. This mechanism has been implemented in major browsers already during their early versions [33]. For this reason, it is practical to implement and recommended from the security point of view.

The second security issue from HTTP-level vulnerabilities is XSS. This is related with JSON and reading files from API relying on this format. That is why it is also called JSON vulnerability. Only in a scenarios when JSON content is interpreted by vulnerable processor it is possible to override native JavaScript object constructors. Prevention includes disabling this script from abilities to be executed by starting JSON response with `"}]}',\n"` or answer by only POST requests. Angular provides built-in protection by recognizing the `"}]}',\n"` string and stripping it off before parsing incoming responses.

²⁷Angular's module for handling communication services over HTTP.

²⁸However, there is still a risk of Domain Name System (DNS) spoofing attacks. The network layer requires to enable Domain Name System Security Extensions (DNSSEC) in order defend against DNS spoofing attacks.

²⁹Interceptors handles incoming responses and outgoing requests.

³⁰Random data used for modification of encryption to safeguards passwords.

³¹Hash code is equivalent term to hash values and digest — all those are related to output of a hash function.

3.3.2. Programming Best Practices

Angular intensively uses TypeScript. For that reason, best practices for Angular, JavaScript and TypeScript should be combined. Some of those techniques are universal, can and should be practiced in all three environments.

JavaScript basic best practices [34] focus on:

- Minimization of usage: global variables, creating objects using *new* keyword, comparison variables using the double equal sign (`==`) and execution of *eval()* function.
- Local variables should be used as often as possible. Declarations should be placed on the top using *let* or *const* keywords. Important is not to declare variable without those keywords, because they can accidentally overwrite an existing global variable. Declaration of variables should be combined with their initialization.
- The keyword *new* with its desired data types should be replaced with its short version, i.e. `{}`, `""`, `0`, `false`, `[]`, `/()/`, `function(){}.` Adequately, it stands for: creating object, primitive string, primitive number, primitive boolean, array object, regexp object and function. Otherwise, unexpected behaviours might occur. An example could be `var my_arr = new Array(5); console.log(my_arr.toString());` which returns `,,,,,`. Whilst `var my_arr2 = [5]; console.log(my_arr2.toString());` will return what it is expected to return, i.e. `5`.
- Operator to compare values `===` should be used — it compares values and types, while `==` only checks if the values are equal.
- Usage of default parameters is also important, because value of a missing argument is by default assigned to *undefined*. Such slightly mistake might end up with breaking the whole application.
- Not the least — *switch/case* statements should end with a *default* keyword.

JavaScript is a loosely typed programming language. Therefore, automatic type conversions occurs if best practices will not be used. Developer should be aware of this; avoiding overriding variables is crucial in JavaScript world. TypeScript solves this problem due to its strongly typed nature.

TypeScript uses many JavaScript methods, thus best practices for this language described in previous paragraphs apply. However, it has its own ecosystem and as every different programming language has specific best practices [35]. These are:

- TypeScript, as the name suggests, is a strongly typed language. Hence, with definition of variables explicitly defining a type is strongly recommended. Otherwise, it will be assigned implicitly.
- Typing can be achieved using *boolean*, *number*, *object*, *string*. *Boolean*, *Number*, *Object*, *String* are reserved words in JavaScript and cannot be used in TypeScript.
- Functions are supposed to return specific type as well. When specifying return type developers should keep in mind that *any* should not be used in functions with callbacks.

- As a general approach to write secure code developers should not use optional parameters unless it is really needed. The same situation is in TypeScript's optional parameters in callbacks, e.g. in interfaces.
- Overloads³² should be grouped in an order that more specific signatures are before more general overloads³³.
- Writing few overloads for a scenario in which they differ only by one or two parameters should be avoided as much as possible.
- Multiplying lines of code which differs by one optional parameter is against DRY principle. In TypeScript developers can use optional parameters, thus using them should be practiced.
- Union types are preferred in situations when writing overloads which are different by type in only one argument position.

Angular best practices are defined by the company behind this framework, i.e. Google. The instructions to follow in order to achieve clean code in the projects are:

- Specific file structure convention which separated by dots describes name of the class, name of the element and ends by *.css*, *.html*, *.spec.ts* or *.ts*, e.g. *home.component.ts*. Such kind of convention organize Angular application in an ordered way, simplifies maintenance and makes easier to understand what is inside of each file. It influences also how class names should be declared, e.g. *HomeComponent* (*UpperCamelCase* naming) in this case.
- Following the same rule, components selectors³⁴ shall follow *dashed-case*, e.g. *app-home*. Functions preferable are in *lowerCamelCase* syntax, e.g. *uploadFile()*.
- One of SOLID³⁵ principles has been adapted into this framework best practices, i.e. Single Responsibility Principle (SRP). It helps developers to define one functionality per file, limits lines of code for each defined element and prefers small functions. It simplifies easier maintenance, improves readability and testability.
- The Angular team strictly defines when to declare variables with *const*, i.e. when their values are not supposed to change during application lifetime. Other rules to follow is to define services as singletons³⁶, implement lifecycle hooks³⁷ interfaces, separate styles with templates and using directives³⁸ for enhancing HTML elements.

Those good practices include and should be applied into all Angular elements, i.e. components, directives, interfaces, modules, services³⁹, pipes⁴⁰ and testing files [36].

³²Creating multiple methods with the same name, but with different number or argument types.

³³Signatures is equal term to overloads.

³⁴Unique tag for a component used internally within Angular application.

³⁵Acronym for five design principles which aim achieve high quality software.

³⁶Design pattern which aims from class to have only one instance.

³⁷Stages of components lifecycle.

³⁸Custom behaviours added to the HTML syntax in order to extend some specific functionality.

³⁹Objects which are supposed to execute narrow, well-defined functionality. Also known as singleton objects.

⁴⁰Elements used to transform values, e.g. filtering.

More general programming best practices [37] include⁴¹:

- Avoiding deep nesting, obvious comments and reserved words.
- Balanced usage of object oriented and procedural programming.
- Capitalization of Structured Query Language (SQL) keywords.
- Dividing code to logical parts.
- Consistent indentation, naming scheme and temporary names.
- Documentation of source code.
- Following the DRY principle.
- File and folder organization to uniform standard.
- Limiting line length to avoid too long horizontal lines of code.
- Proper commenting in a project.
- Separation of code and data.

3.3.3. Testing

The third important factor to develop a secure application written in Angular apart from security itself and programming best practices, is testing. It applies not only to Angular. In JavaScript environment it can be achieved using many different tools, e.g. AVA, Chai, Cucumber, Cypress, Jasmine, Jest, Mocha, Protractor and Tape. Not all of them test the same thing, i.e. Cypress and Protractor are used for E2E testing, whilst AVA, Chai, Cucumber, Jasmine, Jest, Mocha and Tape are used for unit testing.

Apart from those "natural" choices software engineers can distinguish between different approaches of testing. However, there are also different levels, processes, techniques and types. Black box, dynamic, static and white box testing are examples of a testing approach. Whilst levels of testing levels are acceptance, integration, system or unit testing. It all depends by functionalities which has to be tested. That is why there are certain stages of testing, e.g. alpha, closed/open beta testing. Types of tests would include internationalization, localization testing for a typical web shop. Financial industry would be more strict and security testing is a must. On the other hand governmental websites would include accessibility tests.

The thesis focuses on testing techniques which are related to Angular, JavaScript, security, web development topics, as well present daily in the industry working with Agile. Acceptance and unit tests aim is to impact on a quality of software. TDD is a process with an approach that tests are written first, including minimum amount of code to pass them. Such kind of approach allows refactoring, helps to reduce amount of bugs and improves design of an application. However, it can be used when technical requirements are strictly defined at the beginning of development. To develop the application, a more flexible version

⁴¹These are also part of software quality practices widely used by Digital Enablement department of the KPMG N.V.

— BDD, is used. It is helpful in scenarios, where some functionality has to be written, but only during development it clarifies which components exactly will be used. BDD can be seen kind of TDD, but from a higher level point of view and more flexible.

Testing in Angular's logic is mostly about unit testing, also known as isolated testing and functional testing — sometimes called E2E testing. Code coverage is a common mechanism for developers to verify how much of source code has been covered by tests. Angular provides a tool for it. Developers are able to set a minimal percentage of passed tests to obtain a permission for deployment. For this to work during testing phase in Angular CLI an appropriate flag has to be added to the default command, i.e. `ng test --code-coverage`.

As a part of the SSDLC penetration testing should be included as well. SPA inherits the same security risks as normal web application. Moving back to Figure 6 proposed by Gary R. McGraw [15] it can be noticed that in terms of secure development penetration tests are part of this process. Apart from these, other techniques related to security testing should be considered too, i.e. code analysis, risk-based security tests.

3.4. Workflow

Angular comes with its specific workflow and while extending the application features developers sometimes have to look for some enhancements. Projects which empowers applications for specific functionality are called *dependencies*. Such technique gives to the developers a way to handy manage dependencies. This is mainly used to to install, uninstall and manage across shared projects an efficient workflow to speed up development. Another element to consider, while setting up a SPA project is to choose appropriate tools. These are: bundler, CSS style, compiler tool, flavour of JavaScript, its framework, linter, package manager, task runner and test runner.

Chapter 4

4. Design

The SecDevOps methodology places security as a highest step in the SDLC. SecDevOps prescribes that architecture design should include this factor of software engineering already on an early stage of a project. Nowadays, web applications are able to provide rich front-end to the end users using techniques such as Asynchronous JavaScript and XML (AJAX)⁴², often in conjunction with REST. It provides higher responsiveness from a user point of view, because of the usage of AJAX techniques. The UI improvements however may cause new potential security risks in an application. These techniques are also used in SPA-specific concept which applies to Angular applications. That is why better security assumptions for Angular application must be done with high diligence.

4.1. Application Architecture

The SoC is a design pattern which divides each logical section into a separated section (concern). Such solution helps to follow the DRY principle and creates a structured application. In this research it has been included into project assumptions and implemented. A simplified overview of the architecture of the within this research developed application is presented in Figure 11.

This application consists of four main logical areas:

- **apps** — the place where all application modules, components are merged together and application's core settings (including routing) are done.
- **functions** — placeholder for deployment based on Cloud Functions for Firebase⁴³. It contains also information about dependencies to be installed on the deployment server.
- **libs** — reusable code, most of them are so-called *features*. More technically speaking in Angular one entity of such features is called *FeatureModule*. The others are *CoreModule* and *SharedModule*. Difference between each of them is described in the 4.1.1 Modules section.
- **server** — back-end serverless⁴⁴ logic, server-related security settings and Server-Side Rendering (SSR) configuration.

This is just a basic distinction between application's root (*apps*), *functions*, *server* and reusable code (*libs*). The (*apps*) will import all libraries which are supposed to be used during the application's execution. It will also configure basic settings such as rendering in the right order on the page or routing. There are also located E2E test cases. The *functions* folder contains only basic configuration used for serverless computing. From there all application is rendered through dynamically injected logic with a Cloud Functions for

⁴²Technique to interact with a page and dynamically modifying the page without reloading the page.

⁴³Triggers for events without the need of a custom back-end, also known as serverless framework.

⁴⁴Type of cloud computing with a dynamically managed resources by a cloud provider.

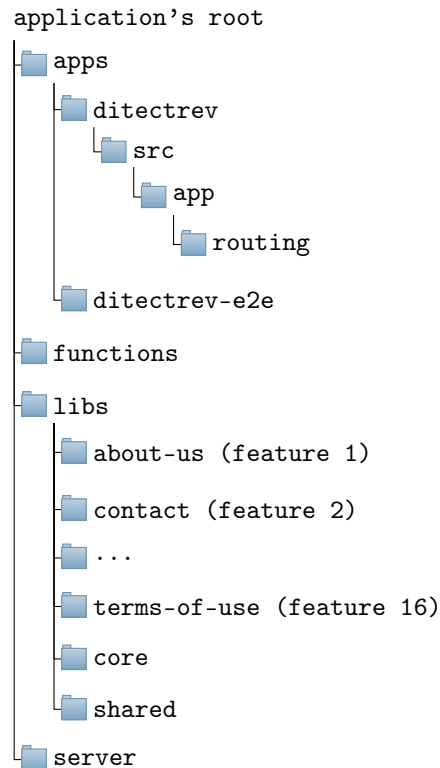


Figure 11: Simplified architecture of the application.

Firebase support. The *libs* will contain most of the source code. It will be mostly so-called *features* which in total in the application are 16. These are application's elements such as contact, home, not found, but also shared components which are on every page, i.e. footer and header. The *server* contains minimalistic back-end, Cloud Functions for Firebase triggers, SSR configuration and some security improvements. On top of this there are also: *core* — the most essential Angular logic to required to run and *shared* — shared code between different part of application.

4.1.1. Modules

Modularity is a software design technique to create separated functionalities of an application into distinct, independent and small units. Modularity has been proven to have positive impact on software design [38]. In Angular applications it is achieved using so-called *modules*. Each module can contain specific logic and metadata. These can be components, directives, exports, imports, services. The logic and metadata has five categories:

1. **bootstrap** — inform Angular from which component the application shall be bootstrapped⁴⁵.
2. **declarations** — components which are available to be used in a particular module.

⁴⁵Automatically starting an application behind the scenes and display this in a browser.

3. **exports** — make available logic from current module in different modules.
4. **imports** — enable exported modules in different modules available in the current module.
5. **providers** — inject services required by other components in the current module. These are directives, services or pipes.

In order to achieve solid SoC techniques exists to create special types of modules. These are about how to split the modules between logical units. Proposed modules include core, features, root, routing and shared. Hence, six kind of modules can be distinguished in an average Angular application:

1. **AppModule** — the only required module. This is the root module that is bootstrapped in order to launch the application.
2. **CoreModule** — modules and services which are declared only once should be implemented here. Examples include module for the HTTP or BrowserModule — containing core application service providers.
3. **FeatureModule** — ordinary Angular module, in most cases a page with separated view. Often lazy loaded to reduce first load time.
4. **RoutingModule** — part of the application to define all URL paths.
5. **SharedModule** — place for components which are used globally in an application, e.g. footer or header. However, it is also the correct place for UI elements. In the application Angular Material UI elements has been imported there as well as other UI libraries.

A module is always a parent for a component. However, they can also contain classes, directives, enums⁴⁶, guards⁴⁷, interfaces, pipes and services.

There are different module systems to be used during development of SPA based on Angular for JavaScript syntax. In the project ES2015 (also known as ES6) has been used. CommonJS has been used in Jest testing environment. ES2015 modules recently became a standard written by ECMA TC39 [39] and for front-end layer are highly recommended to use. Other possibilities includes: AMD, CommonJS, ES3, ES5, ES2015, ESNext, System or UMD [40], e.g. CommonJS modules are widely used in Node.js environment. Whilst AMD often are used as a RequireJS module loader. Further investigation in these topics is out of scope for this research. However, study this topic is highly recommended to fully understand how the modern JavaScript stack works. They are also slightly different in terms of source code analyzers. For example, ES2015 is considered to be easier for them in comparison to AMD and CommonJS [41]. Source code analyzers has been described more in-depth in the 4.2.4 Security Testing section.

⁴⁶Sets of constants.

⁴⁷Interfaces which allows or blocks navigation to a requested route.

4.1.2. Components

In Angular applications the most basic block of an UI is called *component*. Components are defined as a separated elements of a front-end applications and exists across others modern front-end frameworks. Such separated structure simplifies achieving SoC and follow the DRY principle. Each component in Angular consists of four properties:

1. **providers** — Angular's object injected into the specific component which contains certain functionality.
2. **selector** — definition of an unique tag for the component which is reused later in the HTML in order to control which component to display on specific page.
3. **styleUrls** — path to styles of the defined component, alternatively inline styles in the HTML⁴⁸ can be used.
4. **templateUrl** — path to the view, often this is a template file extended by Angular syntax.

Components are kind of equivalents to typical classes known from different programming languages. They also have the keyword *class* in their name definitions and can inherit from interfaces or other classes as well as have its own constructor. Each component has lifecycle hooks, in the order of their execution these are:

- **constructor** — invoked once Angular created a component.
- **ngOnChanges()** — respond each time once change on one of the input properties has been detected and execute assigned logic.
- **ngOnInit()** — initialize the component and set it as ready-to-use.
- **ngDoCheck()** — respond each time there is a change on any event.
- **ngOnDestroy()** — called just before the component is destroyed.

The mentioned hooks are for the component itself, but there are also lifecycle hooks for the components' children. In order of their execution they are the following:

- **ngAfterContentInit()** — invoked after an external content has been loaded.
- **ngAfterContentChecked()** — respond each time after component's content has been checked.
- **ngAfterViewInit()** — invoked after component's view has been initialized.
- **ngAfterViewChecked()** — respond each time when the view of the component has been checked.

⁴⁸This is possible, but is known as one of the CSS anti-patterns. Therefore, it is considered as a bad practice.

4.1.3. State Management

Modern SPA's are often classified as rich web applications. Due to the application's complexity problems may arise when managing states. This is simply due to the reason that currently developed web application have a big number of functionalities, which causes source code to grow. An ordinary used Angular web application consists six types of states [42]:

1. **Client state** — the most basic one on the front-end layer for storing client-side functionality.
2. **Local UI state** — UI components' state on the client-side.
3. **Persistent state** — subset of the server state stored in the client's memory.
4. **Server state** — server-side state, often provided by REST endpoint to the front-end.
5. **The URL router state** — navigation functionality to keep information about which view shall be displayed for the user.
6. **Transient client state** — holding certain actions on the client without metadata in the URL.

The ecosystem of Angular simplifies management of these different states. However, for very large applications the standard state management functionality might not be sufficient. Applications with an enormous codebase might need to introduce more sophisticated libraries. Akita, Apollo, NgRx, NGXS or RxJS can be used to challenge the problem of state management.

4.2. Application Security

The complexity of modern applications is continuously growing. This increased complexity lead to more possible security risks. JavaScript's language-specific flaws are considered to be unsafe and unsecure. However, currently JavaScript does has a strong position compared to other languages in web environment in terms of popularity. Practically, the vast majority of web application are powered by JavaScript, while JavaScript is also gaining popularity in non-web environments (desktop, mobile).

Application security can be broken in many parts. Planning and achieving application security is definitely a broad topic. The nature of JavaScript causes that security is often overlooked. Isolation of the environment (containerization), compilers rules, secure development, security testing may have a positive impact on the security per se. These elements combined with continuous security technique can be seen as an important step to achieve a decent security level of the application.

4.2.1. Containerization

Isolating the development environment from the physical machine has a significant security impact. Virtualization within working environment has been used already for years. Recently many JavaScript packages have been infected [43, 44, 45, 46], which clearly shows that even developers can be victims of hacking activity which might affect the end users.

These packages are widely used dependencies for development, thus the victims were software developers. As this example clearly shows not only people unfamiliar with IT can be a target of an attack, but also the software engineers working with world's largest hub of packages for software industry.

Containerization helps to minimize likelihood of such kind of problems if any dependency would be infected. This is because everything is happening in the *container*, not directly on the working machine. A container is an isolated user-space instance based on minimalistic Linux or Windows distribution [47]. Inside this container is an *image*. Software is loaded into the container with the project's dependencies. This is a kind of virtualization from the development environment where the source code is actually compiled. Each of the container image instances is using the host operating system through container runtime (Docker) as shown on Figure 12.

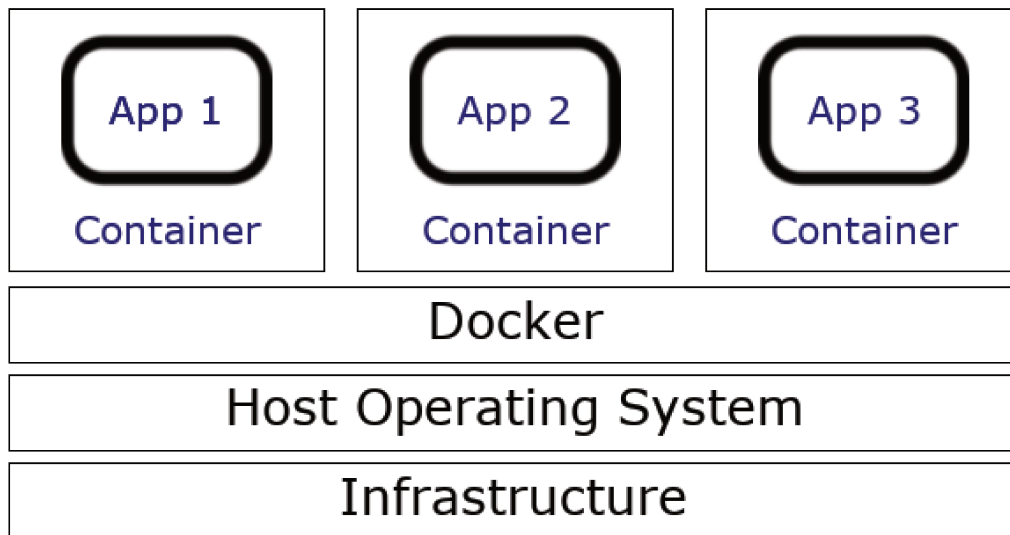


Figure 12: Docker application containers.

Such kind of solution is the reason why containerization is more performant and requires less resources than virtualization. This concept is not new, containerization can be seen as a different type of managing virtualization. From one side they provide an isolated and secure environment. From the other side, they are lighter than virtual machines. Other advantages of using application containers are immutability, portability and scalability [48].

Containers are also often used in the CI pipelines, where each stage has different container. They also simplify working with legacy projects. This is because developers can do everything in a container, i.e. they do not need to install dependencies on their local machines. Often the versions between the time a legacy code has been written and ran again are significantly different. Hence, keeping such project in a container solves the problem of

incompatibility in dependencies versioning when working with legacy projects.

4.2.2. Compilers Rules

There are two types of compilers used in this project, one for compiling TypeScript and another one for Angular. Nx provides support for created components to have different settings on each single scope with a global compilers configuration file in an application root. The project of this application has one file for its global configuration *.tsconfig.json*. In this file Angular and TypeScript compiler settings are defined.

Each of these separately defined modules (under *libs* and *src* contain actually three different compiler setting files as shown on Figure 11). Considering an example one of *libs* modules (e.g. *home*), the structure is as follows:

- **tsconfig.json** — general specification of compilers settings for particular local scope.
- **tsconfig.lib.json** — settings of compilers which apply only to the development environment.
- **tsconfig.spec.json** — settings of compilers which apply only to the testing environment, i.e. unit testing using Jest.

For *apps/ditectrev* the only difference is name of a file for the second settings, i.e. it is *tsconfig.app.json*. This is a place for development settings of application's root components where the page is actually rendered. E2E tests (*ditectrev-e2e*) have *tsconfig.json* and *tsconfig.e2e.json*. The second one works adequately to *tsconfig.app.json*. The *server* module has only *tsconfig.json*. The *functions* is without compiler settings, as it is used only for deployment and testing transpiled project. Apart from that, TypeScript compiler settings have been defined in global *tsconfig.json*, from which all other files mentioned in this section inherits. In case of a conflict for the same rule, the locally defined rules overwrites the global settings.

Default configuration for TypeScript compiler is not that strict in terms of security. For that, several options in TypeScript compiler in a JSON format has to be enabled. These are specified in Listing 1.

```

1 {
2   "compilerOptions": {
3     "alwaysStrict": true,
4     "extendedDiagnostics": true,
5     "noFallthroughCasesInSwitch": true,
6     "noImplicitAny": true,
7     "noImplicitThis": true,
8     "noImplicitReturns": true,
9     "noUnusedLocals": true,
10    "noUnusedParameters": true,
11    "strict": true,
12    "strictBindCallApply": true,

```

```

13     "strictFunctionTypes": true,
14     "strictNullChecks": true,
15     "strictPropertyInitialization": true
16   }
17 }

```

Listing 1: TypeScript's compiler custom stricter rules.

All of them are disabled by default [40] and might have positive impact on security of the application if turned on. Higher code quality is also achieved by following these rules. Explanation on the benefits to enable them for each option is given below:

- **alwaysStrict** — enable parsing strict mode with emitting *"use strict"* for each file.
- **extendedDiagnostics** — show more in-depth diagnostic information.
- **noFallthroughCasesInSwitch** — report error when a *switch/case* would fail.
- **noImplicitAny** — disallow declarations and expressions with implied *any* type.
- **noImplicitThis** — raise an error when *this* has implied *any* type.
- **noImplicitReturns** — throw an error when not all code paths in method returns a value.
- **noUnusedLocals** — enable reporting an error when local variable would not be used.
- **noUnusedParameters** — enable reporting an error if parameter of a method would not be used.
- **strict** — all strict type checking options will be enabled.
- **strictBindCallApply** — stricter checking for *apply*⁴⁹, *bind*⁵⁰ and *call*⁵¹ methods.
- **strictFunctionTypes** — disable bivariant parameter checking for function types.
- **strictNullChecks** — allow *null* and *undefined* types to be used only with themselves and *any* type.
- **strictPropertyInitialization** — ensure *non-undefined* class properties are initialized in the constructor.

The option *strict* in terms of security is crucial. By settings this to *true* the application will enable seven strict type-checkings, i.e. *alwaysStrict*, *noImplicitAny*, *noImplicitThis*, *strictBindCallApply*, *strictFunctionTypes*, *strictNullChecks* and *strictPropertyInitialization*. That way, most of these from Listing 1 will be enabled by this single line.

Angular's compiler rules are defined in a different file, i.e. global *tsconfig.lib.json*. The others *tsconfig.lib.json* files inherits from it. Listing 2 presents a code snippet for Angular compiler rules.

⁴⁹Built-in JavaScript method for calling a function.

⁵⁰Built-in JavaScript method for creating a function.

⁵¹Built-in JavaScript method for calling a function. The difference to *apply* is only about taking arguments in a different way.

```
1 {  
2   "angularCompilerOptions": {  
3     "annotationsAs": "static fields",  
4     "annotateForClosureCompiler": true,  
5     "disableTypeScriptVersionCheck": false,  
6     "enableLegacyTemplate": false,  
7     "strictInjectionParameters": true,  
8     "strictMetadataEmit": true,  
9     "trace": true  
10  }  
11 }
```

Listing 2: Important rules for Angular's compiler.

These following Angular compiler options gives the following benefits:

- **annotationsAs** — enable more sophisticated tree shaking, e.g. Closure Compiler⁵² techniques.
- **annotateForClosureCompiler** — this flag is required by the Closure Compiler.
- **disableTypeScriptVersionCheck** — TypeScript version must be checked if set to *false*. Setting this to *true* would ignore errors about unsupported TypeScript versions.
- **enableLegacyTemplate** — make impossible to use deprecated *<template>* tag to avoid collisions with element which has the same name in the DOM.
- **strictInjectionParameters** — throw an error for parameters without possible to determine injection types.
- **strictMetadataEmit** — emit errors for metadata, that otherwise would be ignored.
- **trace** — show more information during compiling templates.

4.2.3. Secure Development

There are many factors developers should think about during the development phase in order to deliver secure software. Angular offers protection from some of the attacks by default. However, there are also attacks which developers have to take care of such as broken access control, cryptographical protocols weaknesses, Denial of Service (DoS)/Distributed Denial of Service (DDoS), known vulnerabilities, security misconfiguration, server malware and many others. These attacks are not primarily related to the front-end layer. The background of these weaknesses could be hacking, human-prone mistakes, open source vulnerabilities, server-related issues and many others. Therefore, Angular applications developed following the SSDLC should result to be more resistant on typical attacks.

An integral part of secure development is automation of security checks. This not only includes security testing, but code quality checks too. Code smells⁵³ in the application can

⁵²Google's compiler for JavaScript which outputs high quality and well-optimized JavaScript.

⁵³Set of common characteristics which specifies that the source code is not good enough and should be improved.

affect the security itself. A CI pipeline with different type of tests challenges this problem and is presented on Figure 13. The pipeline should have several stages with deployment only if tests are passed. Every change or more precisely every new *git push* to the repository would indicate to run the whole pipeline from start to end. In such a way that with every deployment the likelihood of delivering more secure application is higher.



























| Pipeline Jobs 14 | | | | |
|---|--|--|------------------------|---|
| Status | Job ID | Name | Coverage | |
| Security | | | | |
|  | #280203859 | Dependencies Scanning | 00:12:29 2 days ago |  |
|  | #280203860 | Static Application Security Testing (SAST) | 00:10:46 2 days ago |  |
| Quality | | | | |
|  | #280203862 | Code Quality | 00:14:23 2 days ago |  |
| Test | | | | |
|  | #280203864 | E2E Tests | 00:15:22 2 days ago |  |
|  | #280203863 | Unit Tests | 00:37:46 2 days ago |  |
| Build | | | | |
|  | #280203870 | Build Production | 00:17:57 2 days ago |  |
|  | #280203866 | Build Staging | 00:13:16 2 days ago |  |
| Staging | | | | |
|  | #280203872 | Deploy to Staging | 00:12:27 2 days ago |  |
| Security Staging | | | | |
|  | #280328994 | Security Headers Staging | 00:10:23 2 days ago |  |
| Production | | | | |
|  | #280351906   | Deploy to Production | 00:13:31 2 days ago |  |
| Security Production | | | | |
|  | #280203881   | Security Headers Production | 00:11:18 2 days ago |  |

Figure 13: Multistage GitLab CI pipeline of developed application.

A crucial part of web application security is input validation. For input validation Angular provides security in terms of input patterns for different types of inputs. However, it does not for file uploads — this requires custom implementation by the developer. There

are two main strategies to handle this problem:

1. **Whitelisting** — accept only specific types of files.
2. **Blacklisting** — reject listed files, accept all others.

First of all, in both strategies executable files as well as too large files definitely should not be accepted. List of unsecure file patterns is long, especially including unordinary types of files. For this reason, whitelisting is a better strategy, because the system will accept only these kind of files which are supposed to be accepted. Moreover, the error handling message should be rather general than detailed. By providing a detailed response the attacker might guess what kind of strategy is used.

It is worth noting that even icons might be a potential source of vulnerabilities. The modern standard for icons Scalable Vector Graphics (SVG) relies on a vector graphics rendering. This uses JavaScript for scripting, which means it can be executed by a potential attacker. Data injections and XSS attacks can be mitigated by implementing CSP in combination with Angular's built-in protection.

4.2.4. Security Testing

Protection of software from hypothetical attacks is a primary goal for security testing. Security is becoming an integral part of the SDLC since few years back, but it has been a topic of research for more than 15 years [49]. Seven basic security concepts are authentication, authorization, availability, confidentiality, identification, integrity and non-repudiation. These are evaluated by different types of security tests [50]. The most general approach is to divide them on two categories:

1. **Dynamic Application Security Testing (DAST)** — also known as *black box testing*, is executed in running application where the ethical hacker intentionally introduces fault injections to find weaknesses. In this scenario there is no access to the source code.
2. **Static Application Security Testing (SAST)** — also known as *white box testing*, is performed during development to find security vulnerabilities in the source code without executing it.

These two types of techniques can be done both automatically and manually. It depends on what is supposed to be tested, thus different tools are used for those. The core difference between them are presented in Table 5. A hybrid approach, called *grey box testing* is also sometimes distinguished between these two in which the security is tested by given partial information about the system.

More recent approaches are Interactive Application Security Testing (IAST) and Runtime Application Self-Protection (RASP). The first one is a combination of DAST and SAST which works inside the application. Vulnerabilities can be continuously monitored and identified, because these tests are performed right after the functional tests are done. Due to this fact it can be easily integrated into the CI/CD pipelines. That is why vulnerabilities are detected earlier than using the traditional SAST approach and the unsecure bundle can be automatically dropped from a potential deployment. Another advantage is the code coverage which includes also frameworks and libraries. RASP also works inside the application, but it differs from the IAST. RASP is more like a security tool per se, whilst IAST has

| DAST | SAST |
|--|--|
| Hacker approach | Developer approach |
| Lack of knowledge about used technologies | Access to the source code |
| Finds runtime and environment-related issues | Does not find runtime and environment-related issues |
| Typically used only in web & mobile applications | Supported by almost all kinds of programming languages |
| More expensive to fix | Cheaper to fix |

Table 5: Difference between DAST and SAST [50].

more focus on the security testing itself. RASP is included in the application, analyse it and in case of anomalies sends an issue alert or even blocks an application execution. In such a way it can thwart attacks automatically, especially important for these based on the recently discovered vulnerabilities.

Security testing is an essential part of the SSDLC. Recent research shows that it is easier to implement it in an agile project management frameworks such as Scrum [51, 52]. These findings provide an interesting conclusion. The authors suggests that even the chosen project management framework might influence achieving different security levels during software development. This should be preceded by defining security risks which influence on the decision what should be tested using what kind of techniques and tools.

4.2.5. Continuous Security

Nonstop monitoring of application in terms of security might be a challenging task. The CI/CD apart improving productivity [53, 54, 55] can be helpful in automatic security tests. However, there might be a confusion between CI/CD acronyms. In order to understand the difference between them, it is important to define its meaning [55] and distinguish between these:

- **Continuous Integration (CI)** — integrating code in a shared repository. It builds and tests each commit change automatically⁵⁴. It is part of continuous delivery as well as continuous deployment.
- **Continuous Delivery (CD)** — releasing changes to production as quickly as possible by automatically pushing changes to a staging environment. From a staging based on certain conditions the application can be manually pushed to the production environment.
- **Continuous Deployment (CD or CDE)** — goes one step further than continuous delivery. Deployment to production is fully automated.

Continuous security, similarly to continuous integration, should be an integral part of continuous delivery and continuous deployment. The security checks to achieve decent security level should be strict. Steps to achieve it are usually: container and dependency scanning, DAST and SAST. More sophisticated methods could include: fuzzing, IAST, mutation testing or RASP.

⁵⁴In some scenarios such as editing technical documentation in the source code repository it is not a desired behaviour. Automatic pipeline build can be omitted. An example includes setting up a GitLab CI command which would read Git commits containing information about skipping the build.

From these three the most secure seems to be continuous delivery, where deployment from staging to production is done manually. The reason for that is even though pipelines would pass all tests, then the application might be still not working correctly. Continuous deployment definitely speeds up time-to-production, but also contains some risks [56]. The scenario for this could be for example when providing a wrong API key in the environmental variables of a CI system. Even though locally everything worked correctly, it is always better deploy to staging. The environment is usually almost the same as the production one. Thus, it can be manually tested there and manually deployed to the production once the results of these tests will be positive. That is the reason why continuous delivery seems to be more safe and secure.

4.3. Behavior-Driven Development

Integration of unit tests into Angular development environment is relatively easy due to its component-based architecture. Apart from that, Angular CLI generates the test files automatically when creating a new component. Together with a testing framework it creates a nicely isolated structure of the application literally created for unit testing. Similarly to TDD three main phases are defined: red⁵⁵, green⁵⁶ and refactor⁵⁷. In contrast to the TDD, BDD focuses more on how the application should behave for the end user. TDD is more strictly about implementation of the functionality. The BDD comes from the TDD, but takes more flexible approach in the SDLC: test cases are not that strictly defined and are in a more human-readable format.

Fundamental part of the BDD is Given-When-Then (GWT) approach [57], where

- Given — responsible for setting up a context.
- When — describe an event.
- Then — provide outcomes for the event.

The goal is to provide easy to understand and readable unit tests. By implementing BDD it is easier to have more isolated structure of an application which gives enhancements in terms of security. The code maintenance becomes simpler. Clean separation of application's components is easier for debugging. It improves the chance to detect mistakes during development and to eliminate them before deployment. It can result in higher source code quality and less security flaws in it. The BDD process also helps to focus on user expectations, thus by applying it developers are able to achieve better UX. Developers benefit from the fact that part of the application can be tested without completely compiling and deploying the whole application. Since a small piece of code (unit) is the target it checks if particular part of the application behaves as it is expected.

⁵⁵First phase of BDD/TDD when a test is failing.

⁵⁶Improving unit test to pass the test case.

⁵⁷Possible improvements of a unit test.

Chapter 5

5. Implementation

Awareness of recent cybercrime activities is already a good first step to develop secure application. Identifying security risks and implementing security measures for developed scenario is the next one. Architecture designed from a security point of view is an important factor of the application [58]. Implementation of architecture, clean code, the DRY principle, programming best practices, SecDevOps and overall SSDLC process is challenging and yet another step. Carefully selected technologies with appropriate tools may be helpful in that stage. With security in mind, developers have to find trade-offs between modern techniques and mature solutions. Modern techniques have a newer point of view to resolve some kind of problems. However, mature solutions have been used for already some time and are considered to be more stable.

5.1. Functionalities

The application build as part of this research contains some functionalities typical for a modern web application. It has several views which contain content with associated logic behind. These are typical web application features such as input field complete with input validation, interactive elements, small back-end and routing functionality. Other notable functionalities are:

- **Accessibility** — the application supports assistive technology for users with disabilities. It has been achieved using Agastya⁵⁸, semantic HTML and Web Accessibility Initiative's Accessible Rich Internet Applications specification (WAI-ARIA)⁵⁹ [59].
- **Analytics** — every modern application tracks the user behaviour, the application also contains it. Google Analytics and Hotjar have been used as a technologies for this task by using Google Tag Manager.
- **Internationalization** — Google Translate with Oswald Labs Platform APIs provides a way to smoothly integrate into an application automatic translation in more than 100 languages. It has been achieved implementing the Agastya software into the application.
- **Offline** — a service worker⁶⁰ is part of the application to include PWA characteristics.
- **Responsiveness** — Responsive Web Design (RWD) has been implemented using modern techniques of CSS such as Flexbox Layout and Grid Layout. More precisely, its implementation in forms of directives called *Angular-Flex Layout* has been used.
- **Rich front-end** — significant number of integrations has been performed in order to provide appropriate UX. Many libraries and plugins in a form of external dependencies have been used to achieve this effect.

⁵⁸Software for improving websites' accessibility.

⁵⁹Technology which solves problem of reading web content for people with disabilities by adding special directives to the source code.

⁶⁰Script that runs in a background without interacting with a user.

Visualization of the application is attached to the thesis in the appendix A UI Design Elements. Some of these functionalities can be noticed from these screenshots. seen from the attached screenshots can be found in UI design elements.

5.2. Technology Stack

The application has been build using wide spectrum of technologies. They have been used from the development stage, through testing, ending up on production. These are the following:

- **Core Framework** — fundamental part of this project is Angular with its dedicated CLI. It consists of its own API with number of useful methods for animations, compiling process, dependency injection, forms, handling HTTP, routing, service workers, testing, validators and more. This framework supports most recent major browsers as of the time of writing this thesis [60].
- **Cypress** — efficient test runner for automated E2E tests, which can be used as all-in-one platform instead of few E2E tools.
- **Dependencies** — external dependencies installed during development in order to enrich the application functionalities or development experience. These can be code quality checks, external libraries, security testing tools or UI improvements.
- **Docker** — software for running applications in an isolated environment (container).
- **Firebase** — development platform for mobile and web projects. The developed application has been deployed to Firebase Hosting which is supported by Google Cloud Platform (GCP)⁶¹. Other Firebase features which has been used includes: Cloud Firestore⁶², Cloud Functions for Firebase and Cloud Storage for Firebase⁶³. These helped to create the application in a serverless technology, Firebase Security Rules⁶⁴ was responsible for improving security.
- **Git** — distributed VCS widely used for source code versioning and tracking changes of code repository. Changes are send through HTTPS or Secure Shell (SSH) to source code repository where the project is stored.
- **GitLab** — source code repository which embeds Git and provides an ecosystem for DevOps lifecycle. Contains features such CI/CD integrations, code analysis, container registry, planning tool, project management, security, Source Code Management (SCM) and more. It also provides a secure way to store application secrets, such as passwords, secret API keys and other sensitive properties in a form of environmental variables.
- **Jest** — efficient platform used for unit testing. At this point of time this is currently one of the TDD/BDD frameworks which integrates with Angular seamlessly. The reason to choose Jest is its perfectly matching construction to monolithic repositories.

⁶¹Cloud computing services offered by Google.

⁶²Non relational (NoSQL) database for Firebase.

⁶³Storage service offered by Google within Firebase.

⁶⁴Set of security rules for Firebase databases.

- **NestJS** — a Node.js framework for building back-end web applications. It has been used to set up a minimalistic back-end with security configuration.
- **Nx** — extension for the Angular CLI for better management of Angular-based large projects. It comes with specific architecture which promotes clean code, consistency, productivity and safety. This is equipped with code formatter, easier state management, simplified data persistence, static code analysis, visualization of dependencies in a form of a graph and more useful features helpful during development.
- **RxJS** — library for reactive programming which simplifies asynchronous programming. Functionalities such as filtering through glossary results and subscription for a newsletter were the use cases in this project.
- **Webpack** — module bundler for JavaScript. Transcompiles Angular and TypeScript code into JavaScript. Webpack also enables to work with yet unsupported natively new standards of ECMAScript and translate it into executable by the browsers JavaScript. Other customizations are also possible, mainly based on self-written scripts with corresponding plugins.

5.3. Improving Security

There are several well known techniques to increase security in web environments. The first one is to enable HTTPS on the domain of the web application. This common practice protects against Man-In-The-Middle (MITM) attacks⁶⁵. In that case only breaking encryption or gathering cryptographical secrets seems to be the only way for an attacker to break the system.

However, there are still more techniques to perform a MITM attack. It applies even to websites served over HTTPS unless user does not has installed plugins like HTTPS Everywhere⁶⁶. In one of the possible scenarios an attacker could make a request to a real website over HTTPS and forward it to the victim over unencrypted HTTP. To mitigate this issue HTTP Strict Transport Security (HSTS) policy mechanism is used. The HSTS header forbids the browser to make calls to a known website without Secure Socket Layer (SSL)/TLS. It does works only for users who visited the website. In order to have HSTS policy mechanism working for users visiting the website for the first time a *preload* flag is required. By doing this a SSL/TLS stripping⁶⁷ is mitigated.

Nowadays, several countermeasures are provided by default from some cloud hosting companies. For example, Firebase which is being used in this research to host the application has some of them. The CDN provided by a Firebase partner includes basic (D)DoS server protections. A reverse proxy with a 10 MB payload size is also in place for the serverless back-end. Cloud Functions for Firebase also have rate, resource and time limits by default. Logging and monitoring is also provided in the administrator dashboard. On the other side, Angular provides data sanitization for injection attacks as well as CSRF and XSS protection on the client-side. However, there are still defense mechanisms that developers have to take into account in the development stage.

⁶⁵Type of attack in which an attacker is between two communication parties, reads and possible alter the communication between them.

⁶⁶Extensions for major browsers which serves websites over HTTPS only.

⁶⁷Type of MITM attack that forces a victim's browser to communicate over unencrypted HTTP.

5.3.1. Input Validation

Input fields from HTML need to be validated. Angular has built-in support for input validation. It can be easily achieved what kind of input it is required from the user as Listing 3 shows.

```
1 Validators.email; // Accept only e-mail pattern.
2 FileValidator.maxContentSize(20971520); // Limit size of files to 20 MB.
3 Validators.maxLength(512); // Allow maximum 512 characters.
4 Validators.minLength(2); // Require at least 2 characters.
5 Validators.pattern('[0-9]*$'); // Accept only numbers.
6 Validators.required; // Mark this field as required
```

Listing 3: Input validators in Angular.

These rules are quite straightforward. The developer only has to decide which validator will be applied to a particular input field. This decreases the risk compared to a manual implementation. Custom input validation implementations in a form of regular expressions are considered to be complicated. Wrongly implemented regular expressions could compromise the server.

5.3.2. File Upload Attack Prevention

Unfortunately, Angular does not provide validators for file upload type. The HTML side should whitelist accepted formats like on Listing 4. This in combination to Angular's file upload size validators is a current solution for defining types of accepted files for upload.

```
1 <ngx-mat-file-input [accept]="['.doc', '.docx', '.jpg', '.jpeg', '.pdf', '.png', '.xls', '.xlsx']" (change)="uploadFile($event)" formControlName="fileUploader" multiple type="file">
```

Listing 4: Cloud Storage for Firebase security rules for file upload.

Such validation is easy to bypass. The file after all is uploaded to a cloud storage and this is where a proper validation is required. Listing 5 shows the back-end validation. The back-end validation cannot be omitted. Accepted files are limited to accepted formats using Multipurpose Internet Mail Extensions (MIME)⁶⁸ content types and a designated amount of files size. In such a way unrestricted file upload is handled on front-end and back-end side with limiting the size. Limiting file types decreases the risk of executing dangerous types of files which might lead to execution of malicious scripts on the server. Setting boundaries of file size protects from DoS due to large file upload.

```
1 // Allow write files Firebase Storage, only if:
2 // 1) File is no more than 20 MB
3 // 2) Content type is in one of the following formats: .doc, .docx, .jpg, .jpeg, .pdf, .png, .xls, .xlsx.
4 allow write: if request.resource.size <= 20 * 1024 * 1024
5     && (request.resource.contentType.matches('application/msword')
6         || request.resource.contentType.matches('application/vnd.openxmlformats-officedocument.wordprocessingml.document')
7         || request.resource.contentType.matches('image/jpg')
8         || request.resource.contentType.matches('image/jpeg')
9         || request.resource.contentType.matches('application/pdf')
10        || request.resource.contentType.matches('image/png')
11        || request.resource.contentType.matches('application/vnd.ms-excel'))
```

⁶⁸Standard for Internet formats.


```

12      || request.resource.contentType.matches('application/vnd.
      openxmlformats-officedocument.spreadsheetml.sheet'))

```

Listing 5: Cloud Storage for Firebase file upload security rules.

5.3.3. Security Headers

An effective way to handle various of security weaknesses within JavaScript-related backends is by using a dependency called *Helmet*. Helmet helps in a relatively easy way to modify many HTTP headers related with security and privacy. Its initialization implies adding protection into seven different attacks. The remaining ones have to be set up manually. Listing 6 clearly presents the logic behind it written in NestJS.

```

1  import * as express from 'express';
2  import { Express } from 'express';
3  const helmet = require('helmet');
4
5  const expressApp: Express = express(); // Create Express instance.
6
7  expressApp.use(helmet()); // Enable Helmet's 7 default middleware protections
   , i.e. dnsPrefetchControl, frameguard, hidePoweredBy, hsts, ieNoOpen,
   noSniff and xssFilter.
8
9  // Preload HTTP Strict Transport Security (HSTS).
10 expressApp.use(
11   helmet.hsts({
12     includeSubDomains: true, // Must be enabled, so "preload" will work.
13     maxAge: 31536000, // In seconds, one year.
14     preload: true
15   })
16 );
17
18 expressApp.use(helmet.permittedCrossDomainPolicies()); // Prevent Adobe Flash
   and Adobe Acrobat from loading content.
19
20 // Enforce to expect Certificate Transparency (CT) for 24 hours.
21 expressApp.use(
22   helmet.expectCt({
23     enforce: true,
24     maxAge: 24 * 60 * 60 // In seconds, regard it for max 24 hours.
25   })
26 );
27
28 // Limit website features by implementing Feature Policy.
29 expressApp.use(
30   helmet.featurePolicy({
31     features: {
32       fullscreen: ["'self'"],
33       payment: ["'none'"],
34       syncXhr: ["'none'"]
35     }
36   })
37 );
38
39 server.use(helmet.noCache()); // Disable client-side caching.
40 server.use(helmet.referrerPolicy({ policy: 'same-origin' })); // Send Referer
   header only for pages on the same origin.

```

Listing 6: Declaration of security headers in NestJS.

Initialization of this (line seven) implies that these following middleware will inform HTTP headers⁶⁹ how the browser should behave. The following behaviour is expected from the browser:

- **dnsPrefetchControl** — disable DNS prefetching in the browsers (sets *X-DNS-Prefetch-Control* to *off*).
- **frameguard** — mitigate clickjacking⁷⁰ attacks (sets *X-Frame-Options* to *SAMEORIGIN*).
- **hidePoweredBy** — hide used on the website technological stack (removes *X-Powered-By*).
- **hsts** — enforce keeping users on HTTPS (turns on *Strict-Transport-Security*).
- **ieNoOpen** — inform Internet Explorer no to execute downloads in a client site's context (sets *X-Download-Options* to *noopen*).
- **noSniff** — prevent browsers from trying to guess (*sniff*) a MIME type (sets *X-Content-Type-Options* to *nosniff*).
- **xssFilter** — prevent reflected XSS attack by (sets *X-XSS-Protection* to *1; mode=block*).

However, there are still a number of headers left which can be set to increase the security:

- **contentSecurityPolicy** — whitelist scripts which can be loaded in an application (sets *Content-Security-Policy*).
- **crossdomain** — prevent handling data across domains (sets *X-Permitted-Cross-Domain-Policies* to *none*). Especially it focus on Adobe Flash and Adobe Acrobat which can load content from other sites.
- **expectCt** — browser will expect Certificate Transparency (CT)⁷¹ from the requested website (sets *Expect-CT*).
- **featurePolicy** — restrict which features the application can use (sets *Feature-Policy*).
- **noCache** — disable browser caching (modifies *Cache-Control*, *Expires*, *Pragma* and *Surrogate-Control*).
- **referrerPolicy** — disable forwarding information about site origin when user moves from one site to another (modifies *Referer* and *Referrer-Policy*). This is a privacy enhancement rather than a security issue.

One option has not been used at all, i.e. *hpkp* (HTTP Public Key Pinning (HPKP)). The new *Expect-CT* header is considered to be more flexible and safer [61]. Both of them mitigate the same attack.

⁶⁹However, if particular browser does not support certain header it will not be enforced.

⁷⁰Tricking victim to click on certain element which attacker controls. This element might be invisible or masked.

⁷¹Technique for auditing and monitoring identity certificates (also known as public key certificate or digital certificate). It detects fake and malicious SSL certificates.

5.3.4. Content Security Policy

Previous listings does not include implementation of CSP. This particular security header itself required more source code than all the others headers. Listing 7 presents CSP which limits allowed external scripts which can be executed in the application. The short comments inform for which integrated technology it is required, whilst the others are self-explanatory.

```

1 import * as express from 'express';
2 import { Express } from 'express';
3 const helmet = require('helmet');
4
5 const expressApp: Express = express(); // Create Express instance.
6
7 expressApp.use(
8   helmet.contentSecurityPolicy({
9     browserSniff: false, // Disable browser sniffing.
10    directives: {
11      baseUri: ["'self'"], // Restricts use of the "<base>" tag to origin (
12                           without subdomains). This directive doesn't use "default-src" as
13                           fallback, thus by default it allows anything.
14      blockAllMixedContent: true, // Prevent loading any assets using HTTP
15                                  when the page is loaded using HTTPS.
16      childSrc: [
17        "'self'", // Default policy for valid sources for web workers and
18                  nested browsing contexts loaded using elements such as "<frame>"
19                  and "<iframe>": allow all content coming from origin (without
20                  subdomains).
21        'https://vars.hotjar.com' // Hotjar.
22      ],
23      connectSrc: [
24        "'self'", // Default policy for restricting the URLs which can be
25                  loaded using script interfaces: allow all content coming from
26                  origin (without subdomains).
27        'https://agastya-version.oswaldlabs.com', // Agastya.
28        'https://firebasestorage.googleapis.com', // Cloud Storage for
29              Firebase.
30        'https://firestore.googleapis.com', // Cloud Firestore.
31        'https://platform-beta.oswaldlabs.com', // Agastya.
32        'https://www.google-analytics.com', // Universal Analytics (Google
33              Analytics).
34        'https://*.hotjar.com:*', // Hotjar.
35        'https://vc.hotjar.io:*', // Hotjar.
36        'wss://*.hotjar.com' // Hotjar.
37      ],
38      defaultSrc: [
39        "'none'" // Default policy for fallback for the other CSP fetch
40                  directives [Link of these: https://developer.mozilla.org/en-US/
41                  docs/Web/HTTP/Headers/Content-Security-Policy/default-src]:
42                  disallows everything.
43      ],
44      fontSrc: [
45        "'self'", // Default policy for specifying valid sources for fonts
46                  loaded using "@font-face": allow all content coming from origin (
47                  without subdomains).
48        'https://fonts.gstatic.com', // Google Fonts.
49        'https://script.hotjar.com' // Hotjar.
50      ],
51      formAction: ["'self'"], // Default policy for restricting the URLs
52                          which can be used as the target of a form submissions from a given
53                          context: allow all content coming from origin (without subdomains).

```

```

    This directive doesn't use "default-src" as fallback, thus by
    default it allows anything.
37 frameAncestors: ["'self'"], // Default policy for specifying valid
    parents that may embed a page using "<frame>", "<iframe>", "<object
    >", "<embed>", or "<applet>". This directive doesn't use "default-
    src" as fallback, thus by default it allows anything. This is
    basically clickjacking protection.
38 frameSrc: [
39     "'self'", // Default policy for specifying valid sources for nested
        browsing contexts loading using elements such as "<frame>" and "<
        iframe>": allow all content coming from origin (without subdomains
        ).
40     'https://agastya-version.oswaldlabs.com', // Agastya.
41     'https://vars.hotjar.com', // Hotjar.
42     'https://www.google.com' // reCAPTCHA.
43 ],
44 imgSrc: [
45     "'self'", // Default policy for specifying valid sources of images and
        favicons: allow all content coming from origin (without
        subdomains).
46     'https://www.google-analytics.com', // Universal Analytics (Google
        Analytics).
47     'https://www.googletagmanager.com', // Google Tag Manager.
48     'https://www.google.com', // reCAPTCHA.
49     'https://script.hotjar.com' // Hotjar.
50 ],
51 manifestSrc: ["'self'"], // Default policy for specifying which manifest
    can be applied to the resource: allow all content coming from
    origin (without subdomains).
52 objectSrc: ["'none'"], // Default policy for specifying valid sources
    for the "<object>", "<embed>", and "<applet>" elements. It also
    influences "pluginType" by disallowing all of them. The "pluginType"
    directive doesn't use "default-src" as fallback, thus by default it
    allows anything.
53 scriptSrc: [
54     "'self'", // Default policy for valid sources for JavaScript: allow
        all content coming from origin (without subdomains).
55     "'unsafe-eval'", // Unsecure, but required due to Angular's SSR.
56     'https://agastya-version.oswaldlabs.com', // Agastya.
57     'https://ditectrev.us15.list-manage.com', // MailChimp.
58     'https://platform.oswaldlabs.com', // Agastya.
59     'https://platform-beta.oswaldlabs.com', // Agastya.
60     'https://script.hotjar.com', // Hotjar.
61     'https://static.hotjar.com', // Hotjar.
62     'https://ssl.google-analytics.com', // Universal Analytics (Google
        Analytics).
63     'https://www.google-analytics.com', // Universal Analytics (Google
        Analytics).
64     'https://www.googletagmanager.com', // Google Tag Manager.
65     'https://www.google.com', // reCAPTCHA.
66     'https://www.gstatic.com' // reCAPTCHA.
67 ],
68 styleSrc: [
69     "'self'", // Default policy for valid sources for stylesheets: allow
        all content coming from origin (without subdomains).
70     "'unsafe-inline'", // Unsecure, but required in order to render
        styles generated by Angular compiler, which on SSR are generated
        as inline styles.
71     'https://fonts.googleapis.com' // Google Fonts.
72 ],

```

```

73     upgradeInsecureRequests: true // Block loading of active/passive
        content over insecure FTP/HTTP by "upgrading" the connection to
        secure SFTP/HTTPS.
74   }
75 })
76 );

```

Listing 7: CSP for developed application.

5.3.5. Others

The application has number of other preventions for explicit security improvements. Some of them has been included in a GitLab pipeline before it is deployed. These include:

- Error handling without revealing error details to the client.
- Importing application modules from a path, thus avoiding loading using variables. It could have originated from user input which might be harmful.
- Limit concurrent requests using Express middleware. It can slow down brute-force attacks significantly, in practice making them useless.
- Logging behaviour of different errors and events which helps in proper monitoring.
- Principle of least privilege for database access.
- Requesting manual interaction (also known as automation prevention, i.e. reCAPTCHA⁷²) for expected paths of bot attacks.
- Running Node.js inside a Docker container as a non-root user which by default runs as *root*.
- Prevention of CSRF attacks by adding *X-XSRF-TOKEN* header only if the *XSRF-TOKEN* cookie was generated on the back-end.
- Static code analysis to catch code security bugs and programming mistakes in an early stage. Embedded in a pipeline, helped to detect and eliminate over a dozen issues. There were two exceptions from the default rules: *no-non-null-assertion*⁷³ and *no-shadowed-variable*⁷⁴. A single mistake in the source code will cause the GitLab pipeline to fail.
- Scanning for known vulnerabilities and eliminating them as fast as possible. During the project development about 1,500 known vulnerabilities have been detected and eliminated. This is an integral part of a CI pipeline. The configuration has been set up strictly, i.e. even one low vulnerability causes a failing deployment.
- Secure cookies and sessions management. This prevents cookies from being transferred through HTTP⁷⁵, it disables cookie forgery⁷⁶ and hides revealed by default in sessions application technology.

⁷²System to validate humans and robots.

⁷³Non-null assertions are required within passing data for some scenarios in a compilers strict mode.

⁷⁴Unit test case specific for mocking one of external dependencies.

⁷⁵Even if the application is served via HTTPS it is possible.

⁷⁶Forging an authentication without actually doing it.

- Storing sensitive data such as application secrets in environmental variables.
- Whitelist API keys for external services and from which domains they are allowed to be called.

There are also different elements which implicitly improves security. These are code formatting, E2E tests, unit tests and following all types of programming best practices.

5.4. Clean Code

An important part of programming best practices is clean code. Good documentation, high readability, naming conventions and proper objects grouping are examples of the core ideas. Developers arguably spend more time on reading the code than writing code. That is why providing a clean working environment is important. It reduces onboarding time for new members, decreases maintenance time and simplifies debugging [62]. Strongly typed languages like TypeScript should take advantage of its nature and use types. Examples of bad and clean code shows Listings 8 and 9.

```

1 // Case 1.
2 var x;
3 x = 5;
4
5 // Case 2.
6 currentDate = new Date();
7
8 // Case 3.
9 private crsxn(renderer) {
10     ico = new Mesh(new IcosahedronGeometry(40, 4), new MeshStandardMaterial({
11         color: new Color('#061371'),
12         emissive: new Color('#3f51b5'),
13         transparent: true,
14         wireframe: true
15     }));
16     this.scene.add(ico);
17     return ico;
18 }
```

Listing 8: Bad code in TypeScript.

```

1 // Case 1.
2 const usersNumber: number = 5;
3
4 // Case 2.
5 public currentDate: Date = new Date();
6
7 // Case 3.
8 public scene: Scene = new Scene(); // Create the scene.
9
10 /**
11  * @description Create scene of this animation.
12  * @param {renderer} - the renderer object to display scenes using WebGL.
13  * @returns {Mesh}
14  */
15 public createScene(renderer): Mesh {
16     const radius: number = 40;
17     const detail: number = 4;
18
19     // Create material object with properties for surfaces with highlights.
```

```

20  const material: MeshStandardMaterial = new MeshStandardMaterial({
21      color: new Color('#061371'), // Color of the material.
22      emissive: new Color('#3f51b5'), // Color of emissive light of the
        material.
23      transparent: true, // Make transparent bacground.
24      wireframe: true // Render geometry as wireframe.
25  });
26
27  const icosphere: Mesh = new Mesh(new IcosahedronGeometry(radius, detail),
        material); // Create the icosahedron geometry.
28  this.scene.add(icosphere); // Add icosahedron geometry to the scene.
29  return icosphere;
30 }

```

Listing 9: Clean code in TypeScript.

These two listings represents part of the source code used in the application. Even though the code will be working for all of these examples, there are several programming flaws. Analysis of the source code quality is based on three cases:

- **Case 1** — uninitialized variable x with the *var* keyword. Lack of initialization of variables for some languages causes that it contains a certain value, but it is unpredictable. It is known to be one of the most frequent programming mistakes resulting with security flaws. Especially, it is present in programming languages for which stack variables are not initialized by default, such as C/C++ [63]. A good practice is to avoid it and declare it explicitly. Another mistake which is strongly related with JavaScript is to use a problematic keyword *var*. It has several problems like scoping, possibility of re-declarations and updating as well as hoisting⁷⁷. Instead, *let* should be considered as a preferred way to declare variables since the time of introducing ES2015. The new syntax later is transpiled to the executable by browsers JavaScript. Variables with constant values should use *const* keyword.
- **Case 2** — implicit modifiers which in TypeScript can be: *private*, *protected* and *public*. They help to limit exposure of classes, functions and variables. It is crucial especially if working with two different languages in the same project. In some languages the default modifiers policy is different. An example could be C#, where every property has the *private* modifier, whilst in TypeScript it is *public*. A real world scenario includes front-end on Angular, React or Vue.js. These can use TypeScript, in Angular it is de facto a standard, with back-end in ASP.NET Core which is using C#. Being explicit about the modifiers solves the problem of remembering what kind of policy is in which language. These can be avoid confusing when working with cross-technological projects.
- **Case 3** — function definition without a stated return type, poorly documented and containing unintuitive names without proper convention. An explicit declaration with correctly defined type which this method has to return is cleaner to understand what to expect from this function. Secondly, in the first example it is not correctly documented, it is especially important whilst working with third party libraries. It helps to understand what kind of behaviour it causes, good documentation can be achieved using JSDoc markup language. Thirdly, the good code listing outsources object properties, which helps to understand the technical context. Lastly, correct names with

⁷⁷JavaScript's default behavior of moving declarations to the top.

convention is much better than shortcuts or combined words without uppercase first letters. Conventions like *lowerCamelCase* are a good approach for variables which are supposed to contain more complex objects.

One of the advantages of TypeScript is having the ability to use a better code intelligence tool due to the static type-checking. In the two listings above in the explicit source code it can be noticed what kind of advantage it gives. Even though the code executed by the browsers is JavaScript, TypeScript helps improving security in the development stage. Potentially made mistakes leading to bugs due to the nature of JavaScript can be largely eliminated using TypeScript. Positive effect of using types have been recently researched with the conclusion that it decreases number of bugs by 15% [64]. Apart from bug-catching abilities, strongly typed languages also simplify maintenance of an application once the project grows and reduces the risk of overwriting properties.

Writing clear source code also minimizes the risk of code smells which at the end affects security of the application. Some general basic principles [62] are:

- Avoid code smells like dead code, framework core modifications, hard-coding, large classes, long conditional statements etc.
- Classes and methods should be not too large, follow the DRY and SRP rules.
- Use easy to follow logic within application.
- Adhere to naming convention recommended by the framework or language, follow it consistently and uniformly.
- Grouping variables by importance, name and prefixes.
- Use meaningful, i.e. intention-revealing, pronounceable names with picking up one word per concept.
- Use proper source code comments in order to understand what each class or method is responsible for.
- Unit tests are written and are an integral part of the codebase.
- Use verbs for function names and nouns for classes and attributes.

5.5. Unit Testing

A fundamental part of BDD is unit testing. Tests should be simple and easy to read for non-technical people. Unit tests also have best practices to follow, one of them is the RITEway[65]. The first four letters are abbreviations of **R**eadable, **I**solated/**I**ntegrated, **T**horough, **E**xplicit. These names clearly indicate in what kind of manner unit tests should be written. An application with correctly written unit tests should follow the RITEway and BDD. High level of test coverage might improve overall security of an application. Often it is considered to have at least 80% in order to say it is a reasonable code coverage [66]. It reduces risk of possible software defects which could occur after deployment without unit tests embedded in the SDLC.

Besides if this is TDD or BDD every unit test should answer five questions [67], such as:

1. What has been tested?

2. What should it do?
3. What was the actual output?
4. What was the expected output?
5. How can the test be reproduced?

Following the RITEway and having answers for these questions helps to write tests with keeping the code to absolute minimum. Isolation and minimization is a sign of a good unit test. The only extension to isolated unit tests are when components are based on external dependencies. Then to test it properly, mocking⁷⁸ technique has to be involved in the process.

A core benefit of implementing unit testing is that the logic cannot be changed accidentally which may cause existing functionality to break. Doing unit tests also helps to improve the design of application's architecture. However, there are also less obvious implications of it. One of them is that tests themselves helps to structure how the application should behave. Moreover, by implementing TDD/BDD developers should be also be more confident about their work.

5.6. Server-Side Rendering

Usually Angular executes source code in the browser, but it is also possible to run Angular code on the server-side using Angular Universal. This technique is called *SSR*, its main benefit is to improve Search Engine Optimization (SEO) and enhance performance, especially on low-powered devices. This requires from developers custom modifications, because by default *Client-Side Rendering* (CSR) is used, differences between SSR and CSR are multiple [68] and are represented by Table 6.

| Step | Description for CSR | Description for SSR |
|------|--|---|
| 0. | Requesting a page | |
| 1. | Server sends response to the browser | Servers sends ready to be rendered HTML response to the browser |
| 2. | Browser downloads the assets (content files, JavaScript) | |
| 3. | Application is executed in the browser | |
| 4. | Page is interactable and viewable | |

Table 6: Steps to load page using CSR and SSR .

The main difference is in the process of rendering page contents. In the SSR approach the page is displayed even before it will be ready to interact. In the CSR Angular makes the application ready, once it is clickable and interactable. This small difference makes different results in page speed tests in favor of SSR which might be important in enterprise applications. The SSR for SPA is currently becoming more popular and has been used.

⁷⁸Creating objects that simulates behaviour of real objects.

5.7. Version Control System

Practical part of this research requires significant amount of source code to write. For this purpose Git has been used as a VCS and a Git repository managed by GitLab⁷⁹ has been set up to simplify development. Git is currently the most popular VCS of modern software development. Semantic commits help to review this repository on later stage. Therefore, convention of seven main behaviours has been kept for source code of the application:

- **Add:** — typical commit for adding new logic into the application.
- **Change:** — something has been changed in the project.
- **Delete:** — certain piece of code has been dropped.
- **Fix:** — some functionality has been repaired.
- **Improve:** — code formatting, documentation, error handling, security or any other improvement.
- **Refactor:** — part of the source code is restructured.
- **Update:** — at least one of application's dependencies has been updated.

As a good practice commits are done often with small changes which helps to standardize versioning. However, there is one more important factor of using repository hosted on GitLab. This is its good integration with GitLab CI, a system for CI/CD development practice with various of tools it provides.

Versioning of the application is achieved using so-called semantic versioning. The general idea behind it is to keep in Git tag numbering MAJOR.MINOR.PATCH [70], where:

- **MAJOR** — the most influential additions, e.g. 1.X.X implies first deployable version.
- **MINOR** — this includes more general changes in the application.
- **PATCH** — a small improvement has been added.

The repository should have been configured with permissions for certain collaborators. In such a way that users making changes will require to have the correct permissions. Making *master* as a protected branch is also required and a good practice.

5.8. Deployment

The application is deployed on Firebase Hosting — service for provided by Google for serving web content. Pipelines have been set up and synchronized with Firebase to provide automatic deployment. There are two implemented types of hosting environments:

1. **Staging**⁸⁰ — used for testing purposes before deployment to production.
2. **Production**⁸¹ — the final destination for deployment of the application.

⁷⁹Available to see at https://gitlab.com/danieldanielecki/thesis_app once permission will be received, last access: 26 August 2019.

⁸⁰<https://thesisapp-dev.firebaseio.com/>, last access: 26 August 2019.

⁸¹<https://thesisapp-16048.firebaseio.com/>, last access: 26 August 2019.

Applications hosted on Firebase by default are equipped with: built-in protection against (D)DoS attacks, Content Delivery Network (CDN)⁸², GZIP compression⁸³, HTTP/2 support⁸⁴, Nginx web server, reverse proxy⁸⁵ and SSL certificate⁸⁶.

Firebase provides its own CLI to deploy an application using a terminal. It also smoothly integrates with a CI system. The code is then transpiled which is described in detail in the next section. Webpack is responsible for minification⁸⁷ and obfuscation⁸⁸ of the source code into a form of ready-to-deploy bundle. This bundle is wrapped in a GitLab container and shipped to the GitLab pipeline. The CI system performs automatic tests and once these are passed sends the container to a staging environment. The staging environment is for manual tests to check if the added logic works as expected in a real environment. It has to be tested by a developer or tester. If approved, then the application is shipped to production by manually informing the CI system about it. The whole communication and server content is through secure connection served over HTTPS or SSH.

5.9. Transpilation

Once the implementation part has been finished the last concept to clarify is how actually the source code is executed. A process called *transpilation* is performed, this is a different than traditional compilation. Compiler translates language from high level programming language to its lower representation, e.g. C# to Microsoft Intermediate Language (MSIL) or Java to Java bytecode. In the transpilation process the written source code is compiled from one programming language into another, but with a similar level of abstraction, e.g. CoffeeScript to JavaScript.

There are different module systems as described in the 4.1.1 Modules section. Nevertheless which one would be specify to work with during development, ES5 must and has been specified as a target platform. Currently, that is the standard compatible with all major browsers [69]. Webpack is the tool which in the following project has been used during this research to bundle this technological stack and perform the last step before deployment to have runnable SPA. More precisely, webpack firstly calls the TypeScript compiler, then the Angular compiler and finally builds the bundle during the build time (using AOT). This is how TypeScript using ES2015 modern techniques are translated from the Angular project into JavaScript interpretable by the browser.

⁸²System of geographically distributed proxy servers working together in order to provide fast delivery of web content.

⁸³Method to improve performance for serving web content faster.

⁸⁴Fully multiplexed version of HTTP network protocol.

⁸⁵Type of proxy server with additional layer of abstraction which forwards clients requests to appropriate back-end server.

⁸⁶Data files which cryptographically allow a secure connection between browser and web server.

⁸⁷Remove unnecessary characters in order to minimize the codebase size.

⁸⁸Making source code more difficult for humans to read and understand.

Chapter 6

6. Security Analysis

The evaluation of security of an application is an integral part of SecDevOps and SSDLC methodologies. Security testing of the application is performed in three different ways. With full access to the source code (SAST/white box testing), without any access to the source code (DAST/black box testing) and with limited access to the source code (gray box testing).

6.1. Static Application Security Testing

First of all the source code of the application developed as part of this research is investigated. Some of the techniques have been automated in a GitLab pipeline presented by Figure 13. Thus, the summary concludes some part of the pipeline as well as manual code review. The derived conclusions are:

- E2E tests could cover more elements of the application. One test has been excluded from the GitLab pipeline due to an unknown incompatibility in the GitLab environment. Locally the test passed.
- Files for webpack's browser and server logic are separated. They could have been merged into one file, because webpack enables to have separated logic for client-side and server-side from a single file.
- High result of coverage in unit tests does not covers all application's logic due to Angular's CLI bug [71]. This bug causes that code coverage report generated by Angular's CLI shows executed code. Adding test cases for the logic which was not executed gives full code coverage of certain component. One test have been excluded, i.e. *not found* component due to problematic testing a Three.js⁸⁹. This graphics library interacts with DOM intensively and causes problems with the testing platform Jest. The reason for that is Jest is using specific type of DOM, i.e. JSDOM⁹⁰.
- Imports for routing have long relative paths which could have been improved by adding Nx logic to handle this. The compiler showed this as an improvement in terms of a code quality.
- Most possible strict compiler rules have been applied.
- No known vulnerabilities were found out of 918.296 dependencies checked.
- Old secrets have been found in the GitLab repository. These have been submitted accidentally when changing hiding secrets strategy. However, the credentials has been changed immediately due to the reason that they will be in the repository forever.
- Static analysis of source code has been passed with two exceptions described in the 5.3.5 Others section, i.e. *no-non-null-assertion* and *no-shadowed-variable*. Security rules for static analysis has been applied.

⁸⁹ JavaScript library for 3D browser graphics.

⁹⁰ Vanilla JavaScript implementation of DOM, which aims to speed up execution of logic placed inside it.

6.2. Dynamic Application Security Testing

There are several methods to check the application's security from a hacker's point of view. These include amongst others information gathering, manual testing and scanning techniques. A comprehensive security analysis is presented in the following sections.

6.2.1. Information Gathering

When it comes to penetration testing the first step is gathering information about the target system. Kali Linux⁹¹ with its penetration tools has been used for this. First of all, port scanning has been performed using nmap⁹². From 1.000 scanned ports, the 80 and 443 ports gave some information. Port 80 showed that allowed HTTP methods are GET, HEAD, POST and OPTIONS. It was determined that the proxy might be redirecting requests. It also detected Varnish — a caching tool. It was revealed that port 443 was served by Nginx as a web server and it was possible to retrieve several cryptographic properties about the SSL certificate. The type of public key used was Rivest–Shamir–Adleman (RSA) encryption with 2048 bits which holds a valid certificate for one year. The signature algorithm used was SHA256withRSAEncryption with HTTPS challenge *tls-alpn*. Most of the certificate information can be also found from a Mozilla Firefox browser. The configuration is the recommended compatibility as of the time of writing this thesis [72]. It did also find out that HTTP/2 was used and a TCP sequence prediction difficulty with a score of 17 was determined. According to nmap documentation [73, 74], the result 17 is hard to break and thus considered to be secure.

The data mining tool Maltego found information about the actual network infrastructure. It showed that the CDN was operated by Fastly Network Operations. It located *ns-cloud-c1.googledomains.com* as the DNS for the domain *thesisapp-16048.firebaseio.com*. Personal data was hidden and showed only business details of Google.

Wappalyzer is a tool to discover technologies used on certain website. The front-end technologies were revealed, but not the back-end (for which Express and Node.js are used). It also did not show information about Nginx as a web server. However, this had been already discovered using nmap.

More investigation has been provided by Nikto. This is a scanner with in-depth analysis about web servers. The interesting logs are presented on Listing 10 and shows some potential findings:

```

1 + The Content-Encoding header is set to "deflate" this may mean
   that the server is vulnerable to the BREACH attack.
2 + /phpEventCalendar/file_upload.php: phpEventCalendar 1.1 and
   prior are vulnerable to file upload bug.
3 + /contents/extensions/asp/1: The IIS system may be vulnerable
   to a DOS, see https://docs.microsoft.com/en-us/security-
   updates/securitybulletins/2002/MS02-018 for details.
4 + OSVDB-4598: /members.asp?SF=%22; alert(223344);function%20x(
   {v%20=%22: Web Wiz Forums ver. 7.01 and below is vulnerable

```

⁹¹Distribution of Linux designed for security testing.

⁹²Network scanner which sends packets and analyzes its responses.

```

    to Cross Site Scripting (XSS)\index{Cross-Site Scripting (
    XSS)}. http://www.cert.org/advisories/CA-2000-02.html.
5 + /servlet/com.unify.servletexec.UploadServlet: This servlet
    allows attackers to upload files to the server.
6 + OSVDB-3233: /index.html.ee: Apache default foreign language
    file found. All default files should be removed from the
    web server\index{Web Server} as they may give an attacker
    additional system information.
7 + OSVDB-3233: /index.html.it: Apache default foreign language
    file found. All default files should be removed from the
    web server\index{Web Server} as they may give an attacker
    additional system information.
8 + /151.101.1.195.tar: Potentially interesting archive/cert file
    found.
9 + /thesisapp-16048firebaseapp.jks: Potentially interesting
    archive/cert file found.
10 + /thesisapp-16048firebaseapp.tgz: Potentially interesting
    archive/cert file found.
11 + /backup.egg: Potentially interesting archive/cert file found.
12 + /thesisapp-16048.firebaseapp.egg: Potentially interesting
    archive/cert file found.
13 + OSVDB-3092: /trafficlog/: This might be interesting...
14 + OSVDB-3092: /user/: This might be interesting...
15 + OSVDB-3092: /users/: This might be interesting...
16 + OSVDB-3092: /webaccess.htm: This might be interesting...
17 + OSVDB-3092: /webaccess/access-options.txt: This might be
    interesting...
18 + OSVDB-3093: /database/metacart.mdb+: This might be
    interesting... has been seen in web logs from an unknown
    scanner.
19 + OSVDB-3093: /OA_JAVA/Oracle/: Oracle Applications portal
    pages found.
20 + OSVDB-3093: /OA_JAVA/servlet.zip: Oracle Applications portal
    pages found.
21 + OSVDB-3093: /OA_JAVA/oracle/forms/registry/Registry.dat:
    Oracle Applications portal pages found.
22 + 7864 requests: 0 error(s) and 109 item(s) reported on remote
    host
23 + End Time: 2019-08-17 23:02:55 (GMT0) (2676 seconds)

```

Listing 10: Nikto logs.

There are several potential interesting files and information about possible vulnerabilities on the server-side. Interesting are the findings about two different web servers, i.e. Apache and Internet Information Services (IIS). Even though the application is served by Nginx, it looks like the Firebase infrastructure uses mixed technologies. This includes also a Java and a PHP stack.

Network security investigation revealed that neither DNSSEC nor firewall is enabled for

the application.

6.2.2. Web Scanners

Another method for security scanning includes online web scanning for typical web vulnerabilities. The application has been tested by several of these, i.e. Checkbot: SEO, Web Speed & Security Tester⁹³, CryptCheck⁹⁴, ImmuniWeb⁹⁵, Mozilla Observatory⁹⁶, Pentest-Tools⁹⁷, Security Headers⁹⁸, SiteCheck⁹⁹, Qualys SSL Labs¹⁰⁰. All of the scans provided excellent results for the developed application. Table 7 presents grading score for all these web scanners.

| Name of a web scanner | Score |
|--|-------------------|
| Checkbot: SEO, Web Speed & Security Tester | 92% |
| CryptCheck | A |
| ImmuniWeb | A |
| Mozilla Observatory | A+ |
| Pentest-Tools | Low security risk |
| Security Headers | A |
| SiteCheck | Low security risk |
| Qualys SSL Labs | A+ |

Table 7: Web scanners results for developed application.

One of them — Mozilla Observatory has been integrated in GitLab pipeline with a minimum result to pass 100 out of 100. Due to some additional security configuration the score actually was above the result of 100. The general score shows Figure 14, whilst more in-depth analysis Figure 15.

⁹³<https://chrome.google.com/webstore/detail/checkbot-seo-web-speed-se/dagohlmhagincbfilmkadjgmdnkjinl?hl=en>, last access: 27 August 2019.

⁹⁴<https://tls.imirhil.fr/https/thesisapp-16048.firebaseio.com>, last access: 27 August 2019.

⁹⁵<https://www.immuniweb.com/websec/?id=0y27vLRH>, last access: 26 August 2019.

⁹⁶<https://observatory.mozilla.org/analyze/thesisapp-16048.firebaseio.com>, last access: 29 August 2019.

⁹⁷<https://pentest-tools.com/home>, last access: 26 August 2019.

⁹⁸<https://securityheaders.com/?q=https://thesisapp-16048.firebaseio.com>, last access: 26 August 2019.

⁹⁹<https://sitecheck.sucuri.net/results/https/thesisapp-16048.firebaseio.com>, last access: 26 August 2019.

¹⁰⁰<https://www.ssllabs.com/ssltest/analyze.html?d=thesisapp-16048.firebaseio.com&s=151.101.1.195>, last access: 26 August 2019.

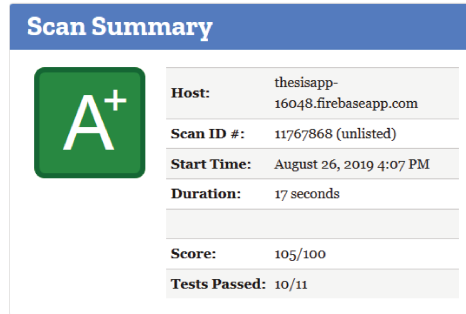


Figure 14: General security analysis score by Mozilla Observatory.

| Test Scores | | | | |
|--|------|-------|---|------|
| Test | Pass | Score | Reason | Info |
| Content Security Policy | ✗ | -10 | Content Security Policy (CSP) implemented, but allows 'unsafe-eval' | ⓘ |
| Cookies | ✓ | +5 | All cookies use the <code>Secure</code> flag, session cookies use the <code>HttpOnly</code> flag, and cross-origin restrictions are in place via the <code>SameSite</code> flag | ⓘ |
| Cross-origin Resource Sharing | ✓ | 0 | Content is visible via cross-origin resource sharing (CORS) files or headers, but is restricted to specific domains | ⓘ |
| HTTP Public Key Pinning | — | 0 | HTTP Public Key Pinning (HPKP) header not implemented (optional) | ⓘ |
| HTTP Strict Transport Security | ✓ | 0 | HTTP Strict Transport Security (HSTS) header set to a minimum of six months (15768000) | ⓘ |
| Redirection | ✓ | 0 | Initial redirection is to HTTPS on same host, final destination is HTTPS | ⓘ |
| Referrer Policy | ✓ | +5 | Referrer-Policy header set to "no-referrer", "same-origin", "strict-origin" OR "strict-origin-when-cross-origin" | ⓘ |
| Subresource Integrity | — | 0 | Subresource Integrity (SRI) not implemented, but all scripts are loaded from a similar origin | ⓘ |
| X-Content-Type-Options | ✓ | 0 | X-Content-Type-Options header set to "nosniff" | ⓘ |
| X-Frame-Options | ✓ | +5 | X-Frame-Options (XFO) implemented via the CSP <code>frame-ancestors</code> directive | ⓘ |
| X-XSS-Protection | ✓ | 0 | X-XSS-Protection header set to "1; mode=block" | ⓘ |

Figure 15: In-depth security analysis by Mozilla Observatory.

The CSP shows *unsafe-eval* and *unsafe-inline* which is caused by Angular's SSR. The first finding can be denoted from the Figure 15, whilst the second in details of the scan by following the link <https://observatory.mozilla.org/analyze/thesisapp-16048.firebaseio.com> (last access: 29 August 2019). When Angular renders page on the server it performs inline styling. Disabling *unsafe-eval* in CSP caused the application did not work properly. Thus, for SSR these two are required to render the application correctly. It clearly shows that for security weakness when using Angular with SSR.

6.2.3. Skipfish

Out of many penetration testing one of them provided unique insights. The reconnaissance tool Skipfish was able to detect many things, which most of the other tools even did not note. Findings presents Figure 16 presents.

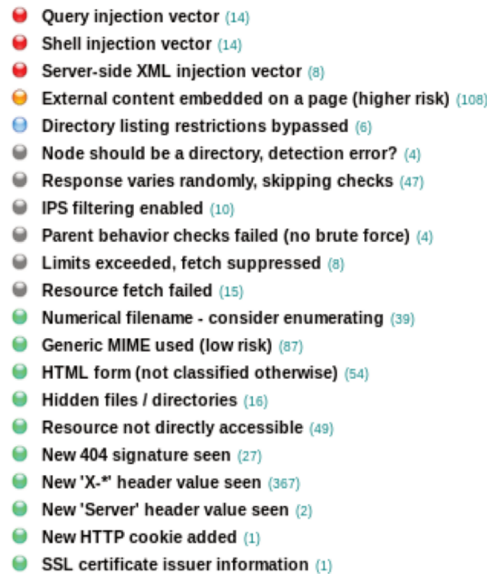


Figure 16: Skipfish results.

The first three which all are related with vector injection are considered to have a high risk. Investigation of these issues revealed niche attack vectors known from fuzzing. This technique provides invalid, random and unexpected inputs. The possible attack vectors had different characters to some paths of the application, often images. These were for example paths to images with additional endings such as `/"`, `'true'` or `/sfish>'>"<sfish><%2Fsish>`. Unknown paths were also part of the result, i.e. `https://thesisapp-16048.firebaseio.com/F.c/"'true'`. This is interesting and niche finding. It clearly shows that there will be always certain attacks which are difficult to find and mitigate. Skipfish was also able to detect several others as seen on the Figure 16. The next serious result is about external content, but this is not surprising since several APIs are used. This is controlled by CSP and should be not harmful. Medium risk contains bypassing directory listing restrictions, whilst all others checks are positive with one exception. This is about failing when fetching a resource, it might be an error in application or in network during the test. It could be because sometimes this attack was stopped due to exceeding rate limit (100 connections per 15 minutes).

6.2.4. Stress Testing

Robustness of software beyond usual limits can determine DoS. SlowHTTPTest¹⁰¹ showed good results for all four types of attacks simulation. Each of the simulation have taken

¹⁰¹Tool for simulation of DoS attacks on an application layer.

different amount of time with 2.000 connections for each of them. During the tests the target server was available for around 98% of the time of test. The attacks simulations were: Apache Killer¹⁰², Slowloris¹⁰³, R-U-Dead-Yet¹⁰⁴ and SLOW READ¹⁰⁵. Apache Killer took the shortest amount of time, i.e. 49 seconds. After that time all connections were closed and server was available for 100% duration of the test. Slowloris took 648 seconds to close all connections. During that test between 279 and 293 seconds the server was not available. SLOW READ took 4683 seconds and no anomalies have been reported. The R-U-Dead-Yet was that one which caused web server unavailable for the longest time. The server was not available for around 26-28 seconds out of 277 seconds of the test. The R-U-Dead-Yet attack simulation is shown in Figure 17.

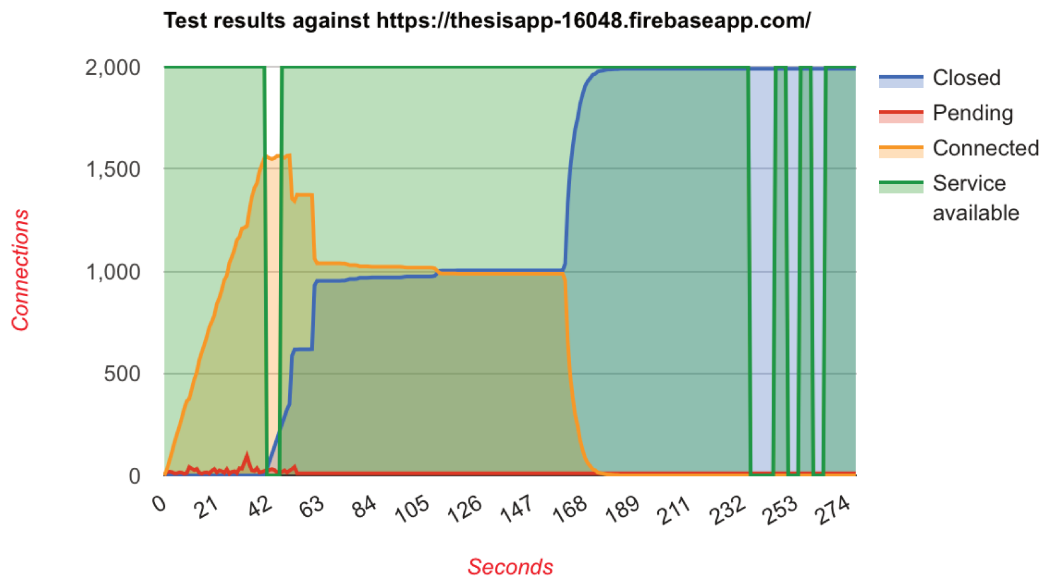


Figure 17: Apache killer attack simulation.

6.2.5. Others

Few other tests have been performed, mostly manual tests. For input fields where human interaction is expected, there exists a bot/automation protection. This includes sending a contact form with a reCAPTCHA check and e-mail confirmation when signing up for a newsletter. When contact form has been sent to the back-end tampering data in HTTP headers was unsuccessful. A tool called HTTrack Website Copier for copying websites has been used. The website was successfully copied and could be used potentially for phishing.

¹⁰²Sending many HTTP GET requests with enormous number of *Byte Ranges* eventually causing running out of memory of the server. This attack is also known as RANGE.

¹⁰³Overwhelming a victim's server by opening and keeping many simultaneous HTTP connections. This attack is also known as SLOW HEADERS

¹⁰⁴Scanning a website for an embedded web forms. This attack also known as SLOW BODY

¹⁰⁵Sending many HTTP requests and reading it very slowly or even not at all.

However, not all functionalities were rendered correctly causing a lack of application's features which for end users are essential. Thus, it could mitigate certain amount of phishing attacks when an attacker would like to use it against a victim.

6.3. Gray Box Testing

Limited access to source code helps to investigate security of particular parts of the application. This sometimes cannot be covered by DAST or SAST. Two testing techniques have been found as valuable for this type of security evaluation.

6.3.1. Cross-Site Request Forgery

The most surprising finding is lack of Angular's protecting against CSRF without creating custom interceptors. This is an open issue in the official repository of this framework and has been not solved yet [75]. Moreover, discussion across specialized forums gave an insights that to handle this problem the official documentation is missing certain steps [76]. On the other side, following these steps would imply setting *httpOnly* to *false* in cookies, which is not recommended. In such a way cookies would be accessible not only for the web server [77]. Thus, implementing security against CSRF would cause lack of security in cookies or vice versa. The finding of CSRF has been confirmed by using Acunetix and OWASP ZAP web application vulnerabilities scanners.

6.3.2. File Upload Attack

Files larger than 20 MB are not accepted as upload. However, the front-end informs the user that combined files cannot be larger than 20 MB, whilst back-end only rejects a single file below 20 MB. For example, uploading a five files, each 19 MB is accepted by the back-end. But sending a form with these attachments together was not accepted. These files are still too small to cause DoS from single file upload. Still this could be development improvement. Files different than DOC, DOCX, JPG, JPEG, PDF, PNG, XLS or XLSX are not accepted by both, front-end and back-end.

6.3.3. API Keys

Many secrets are hidden in the environmental variables. These includes client-side and server-side API keys, credentials and identification data. However, the client side API keys are visible in the HTTP headers. This clearly shows in Figure 18, where the application took the reCAPTCHA API key from environmental variables. Because of that the only one way to limit access when using client-side API keys is to whitelist domains on which it is supposed to be used. The only benefit might be the fact that keeping all API keys in one place simplifies management of external services used across the application.

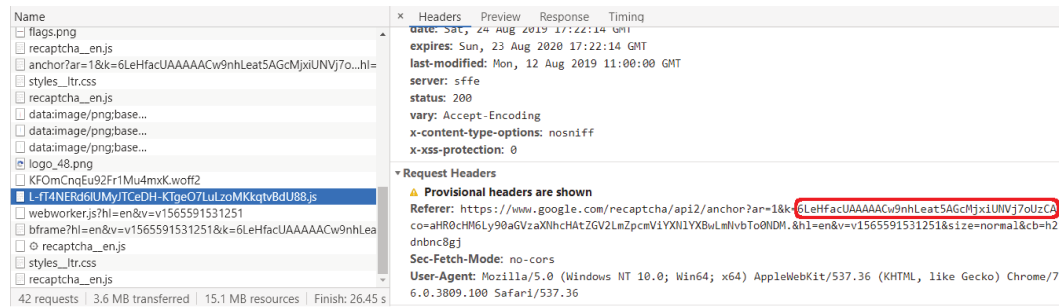


Figure 18: HTTP header with API key for reCAPTCHA.

6.4. Results

Based on the validation results there is no evidence found that the application does not provide a good level of security. Foremost findings are concerns Angular's SSR design which introduction excludes enabling 100% security for CSP. The second unsolved problem is about correct implementation of CSRF. Furthermore there are some niche findings detected by Skipfish which are difficult to mitigate. Definitely from a network perspective lack of DNSSEC and firewall was surprising. Lack of the first one might be due to the reason that the application has been hosted on Firebase domains which are not intended for production. In a real scenario it could have been added when owning a domain. The second one should have been added to the Firebase hosting. SAST showed several improvements which could have been done. It looks like these did not influence the results from outside. Findings from the DAST perspective are more about Firebase infrastructure and cannot be managed by a developers deploying to that platform. The development mistake with revealing secrets when changing hiding secrets strategy could be critical. It has been fixed on time, but for publicly visible repositories it might be a disaster in some scenarios. Having them in private repositories is not unsafe, but secrets should not be kept there. For this, one of correct solutions is to keep them in the form of environmental variables.

Chapter 7

7. Conclusion

The research project provided lots of valuable information about security in SPAs based on Angular. Many techniques have been tested in order to check how to develop an Angular SPA in a secure manner. These methods can be divided in two categories: implicitly and explicitly influencing security. Implicitly influencing security is mainly about delivering high quality code, designing robust architecture, implementing tests or programming best practices. Explicitly influencing security is about improving the security itself. The explicit techniques to improve security are related with detecting compilers rules, hiding secrets, limiting access to database, sensitive data and infrastructure, known vulnerabilities, requesting manual interaction (e.g. reCAPTCHA), setting up security policies or static code analysis. Implicit factors can influence explicit security. However, in this research the impact has been noticed only slightly due to its medium size. For enterprise projects it can be a more important factor. Projects with an enormous codebase can be very fragile. In these kind of applications, the implicit factors are expected to be more influential on the security itself.

Integrating software security in the SDLC of an Angular application is done in many ways. The continuous security approach turned out to be crucial for this, where an important element was automation. The methods for different types of testing and security measurements were automated as much as possible, which assisted in catching mistakes or security vulnerabilities much sooner. However, the research showed that not everything can be automated. That is why manual penetration testing with different types of access to source code should be performed.

Many penetration tools have been used to perform a security analysis. These tools includes Kali Linux tools, online scanners and a few others. The results evaluated the security level of the application as good, however it was still doable to find something niche vulnerabilities. It clearly shows that even if most of the tests passes with a very good results with a proper investigation it is doable to find weaknesses. Finding them is one skill, but exploiting them is another. For these kind of sophisticated attacks there is no easy mitigation. The same applies for vulnerabilities which has not yet been reported. There will be always techniques known by small circle of people which can break application. Thus, security can never be fully guaranteed.

Angular itself provides many built-in protections. However the research showed that the security options which are enabled by the default are not sufficient. Without custom implementation of security defenses such as cookies and sessions management, CSP, security headers and CSRF applications based on Angular should be considered insecure. The SSR implementation for this framework is far from desired in terms of security. Enforcing by design *unsafe-eval* and *unsafe-inline* effectively weakens carefully implemented CSP. Mutual exclusion of cookies security and protection against CSRF should be improved as well. Findings by Skipfish showed niche attack vector which are not easy to mitigate. It clearly shows wide spectrum of possible attacks and how difficult is to build security against it. The framework itself provides a good development environment to develop scalable SPAs, but for the security part there is still a lot to improve which has been shown in the research.

References

- [1] Tim Berners-Lee, *Information Management: A Proposal*, CERN, Switzerland, March 1989.
- [2] Ben Adida, *Helios: Web-based Open-Audit Voting*, Harvard University, 17th USENIX Security Symposium, USENIX Association, pp. 335-348, San Jose, United States, 2008.
- [3] OWASP, *OWASP Top 10 — 2013*, The Ten Most Critical Web Application Security Risks, The OWASP Foundation, 2013.
- [4] OWASP, *OWASP Top 10 — 2017*, The Ten Most Critical Web Application Security Risks, The OWASP Foundation, 2017.
- [5] Mohamed Almorsy, John Grundy, Amani S. Ibrahim, *Automated Software Architecture Security Risk Analysis using Formalized Signatures*, Swinburne University of Technology, ICSE 2013, pp. 662-671, San Francisco, United States, 2013.
- [6] Nenad Medvidovic, Richard N. Taylor, *Software Architecture: Foundations, Theory and Practice*, ICSE'10, pp. 471-472, Cape Town, South Africa, 2010.
- [7] IBM, *How service-oriented architecture (SOA) impacts your IT infrastructure*, January 2008.
- [8] David Sprott, *Business Flexibility Through SOA*, CDBI Forum Limited, 2005.
- [9] Fortune, *Facebook-Cambridge Analytica data breach*, <https://www.bbc.com/news/topics/c81zyn08881t/facebook-cambridge-analytica-data-breach>, last access: 10 January 2019.
- [10] Forbes, *Marriott Breach Exposes Far More Than Just Data*, <https://www.forbes.com/sites/davidvolodzko/2018/12/04/marriott-breach-exposes-far-more-than-just-data/>, last access: 10 January 2019.
- [11] BBC, *Uber Data Breach Exposed Personal Information of 20 Million Users*, <http://fortune.com/2018/04/12/uber-data-breach-security/>, last access: 10 January 2019.
- [12] Techopedia, *Hacking Activities Increase Along with Cryptocurrency Pricing*, <https://www.techopedia.com/hacking-activities-increase-along-with-cryptocurrency-pricing/2/33174>, last access: 10 January 2019.
- [13] European Parliament and of the Council, *Regulation (EU) 2016/679*, Official Journal of the European Union, April 2016.
- [14] Varonis, *60 Must-Know Cybersecurity Statistics for 2019*, <https://www.varonis.com/blog/cybersecurity-statistics/>, last access: 10 January 2019.
- [15] Gary R. McGraw, *Building Security In*, Addison-Wesley, 2006.
- [16] PayPal, *PayPal SOAP API Basics*, <https://developer.paypal.com/docs/classic/api/PayPalSOAPAPIArchitecture/>, last access: 14 January 2019.

- [17] WordPress, *REST API Handbook*, <https://developer.wordpress.org/rest-api/>, last access: 14 January 2019.
- [18] Microsoft, *JSON parsing 10x faster than XML parsing*, <https://blogs.msdn.microsoft.com/sqlserverstorageengine/2017/11/13/json-parsing-10x-faster-than-xml-parsing/>, last access: 3 June 2019.
- [19] OWASP, *REST Security Cheat Sheet*, https://www.owasp.org/index.php/REST_Security_Cheat_Sheet, last access: 14 January 2019.
- [20] F-Secure Labs, *Vulnerabilities*, https://www.f-secure.com/en/web/labs_global/vulnerabilities, last access: 31 January 2019.
- [21] Bernd Grobauer, Elmar Stöcker, Tobias Walloschek, *Understanding Cloud Computing Vulnerabilities*, Siemens, IEEE Security & Privacy, pp. 50-57, March-April 2011.
- [22] NIST, *National Vulnerability Database*, <https://nvd.nist.gov/vuln>, last access: 17 January 2019.
- [23] CVE, *Publicly known Cybersecurity Vulnerabilities*, <https://cve.mitre.org/index.html>, last access: 17 January 2019.
- [24] Thomas S. Beekman, *Auditing Information Security in Mobile Device-enabled Infrastructure*, Erasmus School of Accounting & Assurance Rotterdam, Leiden, Netherlands, 3 October 2016.
- [25] OWASP, *OWASP Proactive Controls*, https://www.owasp.org/index.php/OWASP_Proactive_Controls#tab=OWASP_Proactive_Controls_2016, last access: 10 January 2019.
- [26] Stack Overflow, *Developer Survey Results 2018*, https://insights.stackoverflow.com/survey/2018#most-popular-technologies?utm_source=codecademyblog, last access: 18 January 2018.
- [27] Foundeo Inc., *Content Security Policy Reference*, <https://content-security-policy.com/>, last access: 8 January 2019.
- [28] Google Developers, *Content Security Policy*, <https://developers.google.com/web/fundamentals/security/csp/>, last access: 8 January 2019.
- [29] OWASP, *Cross-Site Request Forgery (CSRF)*, [https://www.owasp.org/index.php/Cross-Site_Request_Forgery_\(CSRF\)](https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)), last access: 8 January 2019.
- [30] OWASP, *Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet*, [https://www.owasp.org/index.php/Cross-Site_Request_Forgery_\(CSRF\)_Prevention_Cheat_Sheet](https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet), last access: 8 January 2019.
- [31] Adam Barth, Collin Jackson, John C. Mitchell, *Robust Defenses for Cross-Site Request Forgery*, Stanford University, Alexandria, United States, 2008.
- [32] Google, *Angular — Security*, <https://angular.io/guide/security>, last access: 9 January 2019.

- [33] Can I use..., *Cross-Origin Resource Sharing*, <https://caniuse.com/#search=cors>, last access: 20 February 2019.
- [34] W3Schools, *JavaScript Best Practices*, https://www.w3schools.com/js/js_best_practices.asp, Refsnes Data, last access: 20 December 2018.
- [35] Microsoft, *Do's and Don'ts*, <https://www.typescriptlang.org/docs/handbook/declaration-files/do-s-and-don-ts.html>, last access: 21 December 2018.
- [36] Google, *Angular — Style Guide*, <https://angular.io/guide/styleguide>, last access: 22 December 2018.
- [37] Envato Tuts+, *Top 15+ Best Practices for Writing Super Readable Code*, <https://code.tutsplus.com/tutorials/top-15-best-practices-for-writing-super-readable-code--net-8118>, last access: 20 February 2019.
- [38] Yuanfang Cai, Ben Hallen, William G. Griswold, Kevin J. Sullivan, *The Structure and Value of Modularity in Software Design*, ACM 2001, pp. 99-108, Vienna, Austria, September 2001.
- [39] Standard ECMA-262, *ECMAScript 2015 Language Specification*, Sections: 15.2.2 Imports and 15.2.3 Exports, Ecma International, Geneva, Switzerland, June 2015.
- [40] Microsoft, *Compiler Options TypeScript*, <https://www.typescriptlang.org/docs/handbook/compiler-options.html>, last access: 5 March 2019.
- [41] Auth0, *JavaScript Module Systems Showdown: CommonJS vs AMD vs ES2015*, <https://auth0.com/blog/javascript-module-systems-showdown/>, last access: 29 May 2019.
- [42] Victor Savkin, *Managing State in Angular Applications*, <https://blog.nrwl.io/managing-state-in-angular-applications-22b75ef5625f>, last access: 8 August 2019.
- [43] The npm Blog, *Details about the event-stream incident*, <https://blog.npmjs.org/post/180565383195/details-about-the-event-stream-incident>, last access: 17 January 2019.
- [44] BleepingComputer.com, *JavaScript Packages Caught Stealing Environment Variables*, <https://www.bleepingcomputer.com/news/security/javascript-packages-caught-stealing-environment-variables/>, last access: 31 May 2019.
- [45] ESLint, *Postmortem for Malicious Packages Published on July 12th, 2018*, <https://eslint.org/blog/2018/07/postmortem-for-malicious-package-publishes>, last access: 31 May 2019.
- [46] The npm Blog, *Reported malicious module: getcookies*, <https://blog.npmjs.org/post/173526807575/reported-malicious-module-getcookies>, last access: 31 May 2019.

- [47] Jurgen Cito, Harald C. Gall, Philipp Leitner, Gerald Schermann, John Erik Wittern, Sali Zumberi, *An Empirical Analysis of the Docker Container Ecosystem on GitHub*, 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), pp. 323-333, Buenos Aires, Argentina, 20-21 May 2017.
- [48] Emily Freeman, *DevOps For Dummies*, John Wiley & Sons Inc., 2019.
- [49] Gary McGraw, Bruce Potter, *Software Security Testing*, IEEE Security & Privacy, pp. 81-85, September-October 2004.
- [50] Synopsys, *SAST vs. DAST: What's the best method for application security testing?*, <https://www.synopsys.com/blogs/software-security/sast-vs-dast-difference/>, last access: 8 August 2019.
- [51] Ihab Mohamed Abdelwahab, Nagy Ramadan Darwish, *A Security Testing Framework for Scrum based Projects*, IJCA, pp. 12-17, March 2016.
- [52] Tomas Klima, Martin Tomanek, *Penetration Testing in Agile Software Development Projects*, IJCIS, pp. 1-7, March 2015.
- [53] Lianping Chen, *Continuous delivery: Huge benefits, but challenges too*, IEEE Software, pp. 50-54, March/April 2015.
- [54] Kent Beck, Mitchell Douglas, Michael Gentili, Tony Savor, Michael Stumm, Laurie Williams, *Continuous Deployment at Facebook and OANDA*, ICSE'16 Companion, pp. 21-30, Austin, United States, 14-22 May 2016.
- [55] Muhammad Ali Babara, Mojtaba Shahina, Liming Zhub, *Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices*, IEEE, pp. 1-32, 2016.
- [56] Muhammad Ali Babar, Adam Johannes Raf, Mojtaba Shahin, Faheem Ullah, Mansoor Zahedi, *Security Support in Continuous Deployment Pipeline*, 2017.
- [57] Robert C. Martin, *The Truth about BDD*, <https://sites.google.com/site/unclebobconsultingllc/the-truth-about-bdd>, last access: 8 August 2019.
- [58] Eduardo Fernandez-Buglioni, *Security Patterns in Practice: Designing Secure Architectures Using Software Patterns*, John Wiley & Sons Inc., 2013.
- [59] World Wide Web Consortium, *Accessible Rich Internet Applications (WAI-ARIA) 1.1*, W3C Recommendation, December 2017.
- [60] Google, *Angular — Browser support*, <https://angular.io/guide/browser-support>, last access: 4 March 2019.
- [61] Threatpost, *Google to Ditch Public Key Pinning in Chrome*, <https://threatpost.com/google-to-ditch-public-key-pinning-in-chrome/128679/>, last access: 8 August 2019.
- [62] Robert C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, Prentice Hall, 2009.

- [63] Common Weakness Enumeration, *CWE-457: Use of Uninitialized Variable*, <https://cwe.mitre.org/data/definitions/457.html>, last access: 19 August 2019.
- [64] Earl T. Barr, Christian Bird, Zheng Gao, *To Type or Not to Type: Quantifying Detectable Bugs in JavaScript*, IEEE, 2017.
- [65] Eric Elliott, *RITeway*, <https://github.com/ericelliott/riteway>, last access: 12 March 2019.
- [66] Yong Woo Kim, *Efficient use of code coverage in large-scale software development*, CASCON '03, pp. 145-155, Toronto, Canada, 6-9 October 2003.
- [67] Eric Elliott, *5 Questions Every Unit Test Must Answer*, <https://medium.com/javascript-scene/what-every-unit-test-needs-f6cd34d9836d>, last access: 12 March 2019.
- [68] Alex Grigoryan, *The Benefits of Server Side Rendering Over Client Side Rendering*, <https://medium.com/walmartlabs/the-benefits-of-server-side-rendering-over-client-side-rendering-5d07ff2cefe8>, last access: 26 August 2019.
- [69] Kangax, *ECMAScript 5 compatibility table*, <http://kangax.github.io/compat-table/es5/>, last access: 4 March 2019.
- [70] OSGi Alliance, *Semantic Versioning*, 6 May 2010.
- [71] Jared Youtsey, *Angular Unit Testing Code-Coverage Lies*, <https://medium.com/ngconf/angular-unit-testing-code-coverage-lies-603c6c85f801>, last access: 29 August 2019.
- [72] Mozilla, *Security/Server Side TLS*, https://wiki.mozilla.org/Security/Server_Side_TLS, last access: 29 August 2019.
- [73] Nmap: the Network Mapper - Free Security Scanner, *Usage and Examples*, <https://nmap.org/book/osdetect-usage.html>, last access: 29 August 2019.
- [74] Nmap: the Network Mapper - Free Security Scanner, *GitHub source code repository*, <https://github.com/nmap/nmap/blob/ac2e140a1483f2c539e71c1aba354d31c8952fc4/ossnmap2.cc#L410>, last access: 29 August 2019.
- [75] Angular - GitHub, *GitHub source code repository open issue: HttpClient does not set X-XSRF-Token on Http Post*, <https://github.com/angular/angular/issues/20511>, last access: 29 August 2019.
- [76] Stack Overflow, *Angular 6 does not add X-XSRF-TOKEN header to http request*, <https://stackoverflow.com/questions/50510998/angular-6-does-not-add-x-xsrf-token-header-to-http-request/50511663>, last access: 29 August 2019.
- [77] CSRF token middleware, *GitHub source code repository: cookie documentation*, <https://github.com/expressjs/csrf#cookie>, last access: 29 August 2019.

Appendices

A. UI Design Elements



IT Consulting Company
Home
Services
Methodology
About us
Contact

Contact

Your name *
Write your name.

Your email *
Provide valid email address.

Your phone number
Your phone number with country prefix.

Project deadline
Specify when at latest you would like to have the project done.

Project description *
Description for your project.

Additional files
Accepted formats: DOC, DOCX, JPG, JPEG, PDF, PNG, XLS and XLSX, maximum files upload size: 20 MB.

Choose preferred form of contact

☐ Email
☐ Phone

Select in which services are you interested in

Cyber Security ☐
Digital Strategy ☐
Software Development ☐

☐ Agree on terms

Send message

IT Consulting Company
Oxford Street 12
+123456789
@ contact@mail.com
Tax ID: 554433

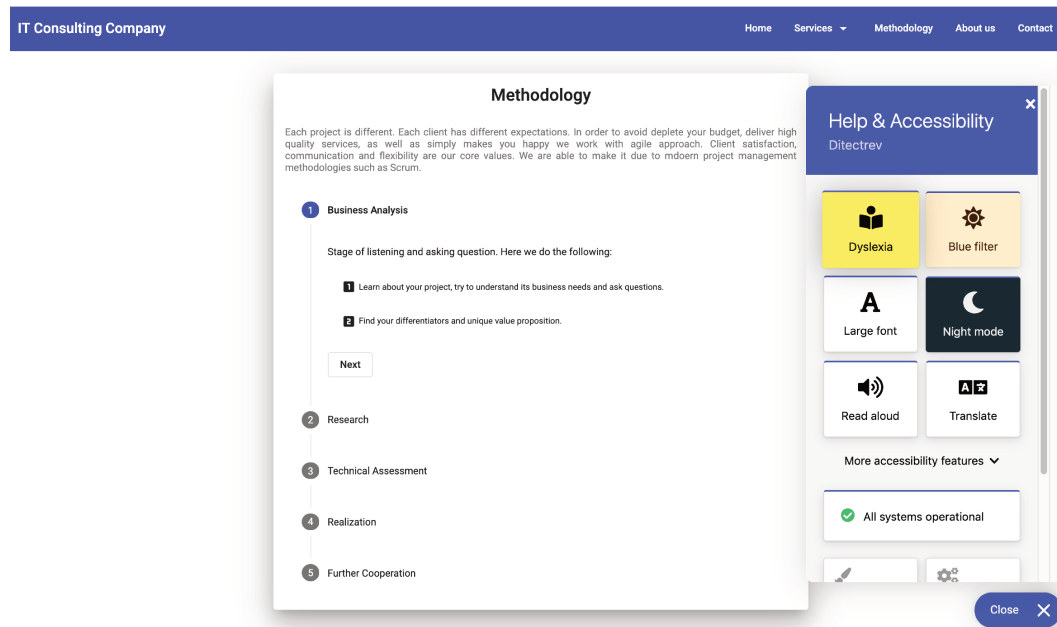
Stay informed
Subscribe
Your email
Email for a newsletter.

Information
FAQ
GLOSSARY
PARTNERSHIPS

Resources
COPYRIGHTS PRIVACY & SECURITY SITEMAP TERMS OF USE

f i in t

© 2019 IT Consulting Company



Index

A

Acunetix, 63
Address Resolution Protocol (ARP), 11
Agastya, 41
Agile, 4, 6, 14, 15, 26, 39
Ahead-of-Time (AoT), 20, 55
Akita, 32
AMD, 30
Angular, 1–4, 7, 8, 17, 19–21, 23–32,
34–37, 40, 42–44, 51, 53, 55, 60,
63–65
 Command Line Interface (CLI), 20,
40, 42, 43, 56
 Flex Layout, 41
 Ivy, 20
AngularJS, 19, 20
Apache, 58
 Cordova, 18
 Killer, 62
Apollo, 32
Application Programming Interface
 (API), 11–13, 20, 21, 23, 41, 42,
61
 key, 12, 13, 40, 42, 50, 63, 64
ASP.NET Core, 51
Atomicity, Consistency, Isolation,
Durability (ACID) compliance,
12
AVA, 26

B

Back-end, 1, 17, 21–23, 28, 29, 41, 43–45,
49, 51, 55, 57, 63
Bash, 18
Behavior-Driven Development (BDD), 3,
8, 27, 40, 42, 52, 53
Bivariant parameter, 35
Botnet, 13
Brute-force, 49

C

C, 18, 51
C++, 18, 51
C#, 17, 18, 51, 55
Cascading Style Sheets (CSS), 1, 18, 25,
27, 41

anti-patterns, 31
Flexbox Layout, 41
Grid Layout, 41
Certificate Transparency (CT), 46
Chai, 26
Checkbot: SEO, Web Speed & Security
Tester, 59
Clean code, 4, 21, 25, 41, 43, 50, 51
Clickjacking, 46
Client-Side Rendering (CSR), 53
Closure Compiler, 36
Code Coverage, 27, 38, 52, 56
CoffeeScript, 55
Common Vulnerabilities and Exposures
(CVE), 13
CommonJS, 30
Containerization, 32, 33
Content Delivery Network (CDN), 43,
55, 57
Content Security Policy (CSP), 20–22,
38, 47, 60, 61, 64, 65
Continuous Delivery (CD), 3, 38, 39, 42,
54
Continuous Deployment (CD or CDE),
39
Continuous Integration (CI), 3, 37–40,
42, 54, 55
Continuous Security, 32, 39, 65
Cookies, 22, 23, 49, 63, 65
 forgery, 49
Countermeasures, 13, 43
Cross-Origin Resource Sharing (CORS),
12, 20, 23
Cross-Site Request Forgery (CSRF), 11,
19, 20, 22, 23, 43, 49, 63–65
Cross-Site Script Inclusion (XSSI), 20, 23
Cross-Site Scripting (XSS), 11, 13,
19–21, 38, 43, 46
CryptCheck, 59
Cryptography, 12, 13, 16, 23, 36, 43, 55,
57
Cucumber, 26
Cyber
 security, i, 1, 4, 9
Cybercrime, 4, 9, 10, 41, 46

Cypress, 26, 42

D

Data Breach, 13

Denial of Service (DoS), 36, 43, 44, 55, 61, 63

Dependencies, 2, 3, 13, 20, 21, 27, 28, 33, 34, 39, 41–43, 45, 46, 49, 53, 54, 56

Dependency injection, 42

DevOps, 15, 16, 42

DevOpsSec, 15, 16

DevSecOps, 15, 16

Distributed Denial of Service (DDoS), 36, 43, 55

Docker, 33, 42, 49
 container, 33, 39, 42, 49, 55
 image, 33

Document Object Model (DOM), 18, 21, 56

Domain Name System (DNS), 11, 46, 57
 spoofing, 23

Domain Name System Security
 Extensions (DNSSEC), 23, 58, 64

Don't Repeat Yourself (DRY), 3, 9, 25, 26, 28, 31, 41, 52

E

ECMA TC39, 30

ECMAScript, 18, 43

Environmental Variable, 40, 42, 50, 63, 64

Error Handling, 12, 38, 49, 54

ES2015, 30, 51, 55

ES3, 30

ES5, 30, 55

ES6, 30

ESNext, 30

Ethernet, 11

Express, 17, 57
 middleware, 49

eXtensible Markup Language (XML), 11, 12

F

File Transfer Protocol (FTP), 11

File Upload Attack, 63
 prevention, 44

Firebase, 42, 43, 54, 55

 Cloud Firestore, 42

 Cloud Functions for Firebase, 28, 29, 42, 43

 Cloud Storage for Firebase, 42, 44

 Command Line Interface (CLI), 55

 domain, 64

 hosting, 42, 54, 64

 infrastructure, 58, 64

Firewall, 58, 64

 rules, 13

Framework, 2, 3, 8, 14, 19–21, 25, 27, 28, 31, 38–40, 42, 43, 52, 63, 65

Front-end, 1–3, 6, 17, 19, 20, 23, 28, 30–32, 36, 41, 44, 51, 57, 63

Fuzzing, 15, 39, 61

G

Git, 3, 39, 42, 54

GitLab, 3, 39, 54, 56

 container, 55

 pipeline, 33, 37–40, 49, 54–56, 59

Go, 17

Google

 Analytics, 41

 Chrome, 3

 Tag Manager, 41

 Translate, 41

Google Cloud Platform (GCP), 42

GZIP compression, 55

H

Hacking, 3, 10, 13, 32, 36

Helmet, 45

Hotjar, 41

HTTP Strict Transport Security (HSTS)
 header, 43

 policy mechanism, 43

HyperText Markup Language (HTML), 2, 18, 21, 22, 25, 31, 41, 44

Hypertext Transfer Protocol (HTTP), 2, 4, 11, 12, 20, 22, 23, 30, 42, 43, 49, 62

 DELETE request, 4

 GET request, 4, 13, 57, 62

 HEAD request, 4, 57

 header, 12, 13, 22, 23, 45, 46, 55, 57, 62–64

- HTTP/2, 55, 57
- OPTIONS request, 57
- PATCH request, 4
- POST request, 4, 13, 22, 23, 57
- Public Key Pinning (HPKP), 46
- PUT request, 4, 13
- request, 13
- response, 23
- Hypertext Transfer Protocol Secure (HTTPS), 11, 12, 42, 43, 46, 49, 55
- challenge, 57
- Everywhere, 43

I

- ImmunWeb, 59
- Immutability, 33
- Incompatibility, 34
- Infrastructure as a Service (IaaS), 13
- Inheritance, 27, 31, 34, 35
- Interactive Application Security Testing (IAST), 38, 39
- Internet Control Message Protocol (ICMP), 11
- Internet Explorer, 46
- Internet Information Services (IIS), 58
- Internet Message Access Protocol (IMAP), 11
- Internet Protocol (IP), 11
- Internet Protocol Suite (TCP/IP), 11
- Ionic, 18
- Isolation, 21, 27, 32, 33, 40, 42, 53

J

- Jasmine, 26
- Java, 17, 18, 55, 58
- Java bytecode, 55
- JavaScript, 1, 2, 17–20, 22–24, 26, 27, 30, 32, 35, 36, 38, 43, 51–53, 55, 56
- JavaScript Object Notation (JSON), 11, 12, 22, 23, 34
- JSON Web Tokens (JWT), 12
- Jest, 26, 30, 34, 42, 56
- jQuery, 19
- JSDoc, 51
- JSDOM, 56

L

- Linux, 33, 57

- Kali, 57, 65

M

- Maltego, 57
- Malware, 10, 14, 36
- Man-In-The-Middle (MITM)
 - attack, 43
- Medium Access Control (MAC), 11
- Microsoft Edge, 3
- Microsoft Intermediate Language (MSIL), 55
- Middleware, 5, 46
- Mocha, 26
- Modularity, 1, 5, 8, 29
- MongoDB, 17
- MongoDB, Express, Angular, Node.js (MEAN) stack, 17
- Mozilla
 - Firefox, 3, 57
 - Observatory, 59, 60
- Multi-page applications (MPA), 1, 17
- Multipurpose Internet Mail Extensions (MIME), 44, 46

N

- NestJS, 43, 45
- Nginx, 55, 57, 58
- NgRx, 20, 32
- NGXS, 32
- Nikto, 57
- Nmap, 57
- Node.js, 1, 17, 30, 43, 49, 57
- Nx, 34, 43, 56

O

- Obfuscation, 55
- Opera, 3

P

- Pentest-Tools, 59
- Phishing, 14, 62, 63
- PHP, 18, 58
- Platform as a Service (PaaS), 13
- Portability, 9, 33
- Programming Best Practices, 1–3, 16, 24, 26, 41, 50
- Progressive Web Application (PWA), 18, 41
- Protractor, 26
- Python, 18

Q

Qualys SSL Labs, 59

R

R-U-Dead-Yet, 62

React, 3, 17, 19, 51

reCAPTCHA, 49, 62–65

REpresentational State Transfer

(REST), 4, 6, 11, 12, 28, 32

RESTful, 4, 11

RESTful Web Services (RWS), 4

RequireJS, 30

Responsive Web Design (RWD), 41

Reverse Proxy, 43, 55

Rivest–Shamir–Adleman (RSA), 57

Rootkit, 14

Routing, 17, 28, 30, 41, 42, 56

Ruby, 18

Ruby on Rails, 17

Runtime Application Self-Protection

(RASP), 38, 39

RxJS, 19, 32, 43

S

Safari, 3

Scalability, 5, 6, 33

Scalable Vector Graphics (SVG), 38

Scrum, 39

Search Engine Optimization (SEO), 53

SecDevOps, 1, 15, 16, 28, 41, 56

Secure Shell (SSH), 42, 55

Secure Socket Layer (SSL), 43, 46, 55, 57

Secure Software Development Life Cycle

(SSDLC), 1, 2, 14–16, 27, 36,

39, 41, 56

Security

analysis, 1, 2, 14, 56, 57, 60, 65

first approach, 2, 16

header, 13, 45, 47, 65

Headers, 59

improving, 16, 29, 42, 43, 52

misconfiguration, 11, 36

risks, 1–4, 11, 13, 20, 22, 27, 28, 32,

39, 41

testing, 2, 26, 27, 32, 36, 38, 39, 42,

56, 57

web, 11

Separation of Concerns (SoC), 9, 28, 30,

31

Server-Side Rendering (SSR), 28, 29, 53,
60, 64, 65

Serverless, 28, 42, 43

Service Worker, 41, 42

Simple Object Access Protocol (SOAP),
6, 11, 12

Single Responsibility Principle (SRP),
25, 52

Single-Page Application (SPA), 1–4, 17,
19, 20, 27, 28, 30, 53, 55, 65

SiteCheck, 59

Skipfish, 61, 64, 65

SLOW BODY, 62

SLOW HEADERS, 62

SLOW READ, 62

SlowHTTPTest, 61

Slowloris, 62

Software

architecture, 4, 7, 28

client-server, 4

component-based, 4, 8, 9, 20, 40

Model-View-Controller (MVC), 4,
6, 7, 20

Model-View-ViewModel
(MVVM), 4, 7, 8, 20

multitier, 4–7

Service-Oriented Architecture
(SOA), 1, 4, 6

craftmanship, 1, 11

development, i, 1, 4, 6, 14, 39, 54

testing

black box, 26, 38, 56

Dynamic Application Security
Testing (DAST), 38, 39, 56, 57,
63, 64

End-to-End (E2E), 3, 26–28, 34,
42, 50, 56

gray box, 56

mutation, 15, 39

penetration, 1, 3, 10, 27, 57, 61

Static Application Security
Testing (SAST), 38, 39, 56, 63,
64

stress, 61

unit, 21, 26, 27, 34, 40, 42, 49, 50,
52, 53, 56

white box, 26, 38, 56

Software as a Service (SaaS), 13

Software Development Life Cycle
(SDLC), 1, 2, 14–16, 28, 38, 40,
52, 65

Spam, 14

Spyware, 14

Static Code Analysis, 3, 16, 43, 49, 65

Structured Query Language (SQL), 18,
26
non SQL (NoSQL), 42

T

Tape, 26

Test-Driven Development (TDD), 8, 26,
27, 40, 42, 52, 53

The Open Web Application Security
Project (OWASP), 3, 11, 12, 14,
16, 22

ZAP, 63

Threat, 4, 13, 16

Three.js, 56

Transmission Control Protocol (TCP),
11, 57

Transpilation, 19, 34, 43, 51, 55

Transport Layer Security (TLS), 22, 43

Trojan horse, 14

Tuple, 19

TypeScript, 2, 18–20, 24, 25, 34, 36, 43,
50–52, 55

U

UMD, 30

Underscore.js, 19

User Datagram Protocol (UDP), 11

User Experience (UX), 2, 8, 17, 40, 41

User Interface (UI), 2, 8, 28, 30–32, 42

V

Validation, 6, 11–14, 20, 37, 42, 44, 49,
64

back-end, 44

file upload, 44

input, 12, 14, 37, 41, 44

Varnish, 57

Version Control System (VCS), 3, 42, 54

Virtualization, 32, 33

Virus, 14

Vue.js, 3, 17, 19, 51

Vulnerabilities, 13, 16, 19–23, 36, 38, 39,
49, 58, 59, 65
Known Vulnerabilities, 3, 11, 36, 49,
56, 65

W

Wappalyzer, 57

Web Accessibility Initiative's Accessible
Rich Internet Applications
specification (WAI-ARIA), 41

Web Application, 1, 2, 4, 5, 11–14,
17–22, 27, 28, 32, 37, 41, 43

Web Server, 4, 17, 22, 55, 57, 58, 62

webpack, 19, 43, 55, 56

Windows, 33

WordPress, 12

X

XML External Entities (XXE), 11

Y

YAML Ain't Markup Language (YAML),
11