Web And Mobile Application Portfolio

Ernesto Venzano Rodriguez

Haaga-Helia University of Applied Sciences

Bachelor's Thesis

2022

Bachelor of Business Administration

Abstract

Author
Ernesto Venzano Rodriguez

Degree
Bachelor of Business Administration

Thesis title
Web And Mobile Application Portfolio

Number of pages
50

This thesis presents a portfolio of software products developed by the author both during their studies in Haaga-Helia and independently. The products consist of two Full-Stack web applications and two built for mobile devices. Development of these projects had been previously started and they were in working condition.

The topic was chosen to support the author's continuing learning in the field of software engineering including both in relation to the technologies used and the best practices for writing high quality applications.

The goals of this project were to research the technologies necessary to implement such products and to present them in a more refined version, explaining also their pourpose and requirements to be met.

This thesis was written over the course of two and a half months in the first half of 2022. The research and writing of the theoretical frameworks took one month and the empirical portion about three weeks. The structure consists of an introduction where the topic and guidelines are defined, one chapter for each product category, each with their own theory and implementation sub-sections, and one last chapter with the conclusions and reflections.

The research was conducted by the author using a varied number of sources, from official documentation for certain products to publications such as books and magazine articles.

As part of the process of writing this thesis, the author has learnt about valuable technologies and practices for writing software which improve the final product and help elevate the quality of the results of the work, as well as improving as a professional. The information contained in this work is useful for students who are also following the path of software engineering as well as those starting their career as programmers.

Keywords
Web development, mobile application framework, iOS development, software engineering, portfolio.

Acknowledgements

My most sincere thanks to Paula for the support and Kasper for the guidance and the feedback throughout this journey. I would also like to acknowledge the help of my peers along these years and even now with their valuable input, and the teachers at Haaga-Helia who provided me the knowledge and the tools to face this task.

To I. & K., for making this the lesser challenge.

Table of contents

1 Introduction

In this thesis, I will present a portfolio consisting of some of my software development projects as examples of Full-Stack Web applications and those for Mobile which could fulfill a client's needs. This project is done as part of the required thesis work and seminar for the bachelor's degree in the BITE programme at Haaga-Helia. The topic was chosen as it is a good fit for the work which I have been doing at school and on my own, following the Software Engineering path of studies. This project will give me the opportunity to show the skills I have learnt during these 3 years and keep improving them based on the market's needs. It will also be a valuable example of my skills which can be used to present my personal projects during job search.

The main structure of this thesis consists of two chapters encompassing the main product categories of this portfolio. The products which will be included in this thesis have been created beforehand, and they are grouped as follows:

- Full-Stack Web Applications, containing two products which were done as pre-assignments for job applications at different Finnish companies. The original assignments are referenced in each project's GitHub repository.
- Mobile Applications; the multiplatform application was done as part of the Mobile Programming course in Haaga-Helia, while the native iOS application was done as part of the Multidisciplinary Software Project course, although solely by me (it was not part of the team project).

In both chapters (2nd and 3rd, respectively), the solutions will be introduced with the theoretical framework. Some of the concepts explained in this sub-section will be the ones required to build, for example, a Full Stack Web Application, including technologies involved, methodologies used and technical or business reasons behind design choices. Afterwards, it will be followed by an empirical part, where the planning and implementation of the solutions are described. In this portion of the thesis, the actual working applications will be presented, along with any refinements done as part of this thesis' work. These software components will reference the repository which holds their source code and will be described in detail with captures of the working implementation.

Finally, there will be a discussion chapter, including a summary of thesis process and how did the journey go, as well as my own reflections on the work done.

Not part of this project are user or client feedback, nor an in-depth description of deployment setups and pipelines, even if the applications themselves might be deployed for internal testing and/or documenting working examples.

The goals for this project are:

- Research the technologies used to build the applications, explaining why they were chosen over alternatives.
- Present the products in a working state, explaining the business needs behind each one of them, how they were developed and implemented, and describing the development journey.
- Improve existing implementations with additional knowledge acquired and feedback received over time, in order to present the best possible solutions.
- Reflect on the work done, noting major achievements and areas for future improvement.

As part of the learning process in this project, the goals are to get knowledgeable with relevant technologies used in commercial projects, as well as best practices of software development, including creating clear, maintainable code.

2 Full-Stack Web Applications

This first section of the portfolio focuses on Web Applications, and more specifically, Full-stack ones. The reason to start here is to bring front and centre these products based on the fact that our studies in Haaga-Helia, when following the Software Engineering study path of the BITE programme, focuses mainly on developing web applications. These first two entries in the portfolio are the ideal example of a student's knowledge and skill set.

2.1 Theoretical Frameworks

The structure of this chapter begins with presenting the technologies used and their theoretical background to introduce the reader into the concepts applied in the developed software.

Web applications are hosted on a web server and accessed by clients remotely, as opposed to traditional applications which reside on the user's computer. The term Full-Stack refers to being in charge of developing both the server and the client software for an application, comprising the whole technical "stack". In a web application, the front end consists of software running on a user's web browser. The back end consists of software running on a server, and it may connect to a database. Thanks to new solutions in cloud provided services, it is actually possible nowadays to have a serverless back end, but this is out of the scope of this thesis. In Full-Stack web applications such as the ones presented here, the front-end client presents the user interface (UI) and sends requests to the back-end web server which process these requests and sends the appropriate responses back to the client. The code which executes between the server receiving the request and sending the response is called middleware, which can be made up of many functions -executing one after the other- and processes the request, including handling any data included with the request and querying the database if needed (Codeacademy 2022).

Both solutions presented in this chapter share the same technical stack. Stacks are often named with an acronym of the underlying technologies. For example, one of the most well-known Web stacks is LAMP, which stands for Linux -OS-, Apache -web server-, MySQL -database- and PHP -scripting language- (Beal 2021). In our case, the stack is **PERN**, which consists of:

- PostgreSQL (Relational Database Management System).
- Express (JavaScript/Node.js back-end framework).
- React (JavaScript library used in front-end development).
- Node.js (JavaScript runtime).

During the development of the apps (even those in the next chapter), management of the code and its changes has been done through version control using Git and hosting on GitHub.

2.1.1 Data Management

Web applications such as those featured in this portfolio have rather complex data management requirements. Although data can be stored in-memory within an application, the ideal solution is to persist the data somewhere. This way, we can assure that it is available to multiple applications or instances of the same application at once, that it will remain accessible even if the application crashes or needs to restart (for example, when applying updates) and the data will be properly managed for future use. In the case of large data sets, it also helps to improve performance, since only the minimum required data can be fetched at once. The best way to do this based on our requirements is using a database. Databases help us solve all those aforementioned problems, as well as enabling us to control the integrity of the data, enforce constraints and processing transactions (Nguyen 2009).

In our courses we have worked extensively with relational databases, with few exceptions (for example, using Google's Firebase on Front-end/Mobile development courses). They are the database type which we have studied in deeper detail and where we can best show our expertise.

Relational Databases are based on the relational model, introduced in 1970 in a paper by E. F. Codd to help solve the problem of multiple arbitrary data structures (Ashdown, Keesling & Kyte 2021, 23). Codd (1970, 379) uses the term relation in the mathematical sense, referring to the relation between elements in different sets. In his paper he also describes concepts such as primary and foreign keys, and the process of *normalization*. In a nutshell, relational databases are called as such since "the data stored in tables can be linked -or related- based on data common to each" (IBM 2019). Relational databases are good for isolating the physical data storage from the logical data structures. The language used to perform tasks on a relational database management system is the Structure Query Language, or SQL. SQL is a declarative language, meaning it is "nonprocedural and describes *what* should be done" (Ashdown, Keesling & Kyte 2021, 27) and commands are written in the form of statements. The reason for this design choice, as pointed out in the paper in which it was presented - "SEQUEL: A Structured English Query Language" by Chamberlin & Boyce (1974, 250)-, is to not limit its use to the technically trained only, making it accessible to a whole new class of users.

The main reason to choose relational databases for these projects, besides the fact that BITE students are best acquainted with them, is that they work very well for general purpose data management. Non-relational, also sometimes referred as NoSQL, are usually ideal when certain specific requirements must be met. These could be, for example, using a document database when data structures are prone to change or a graph database when relationships between entities -nodes- are important (MongoDB 2021).
Some of the most popular relational database engines are Oracle, MySQL, Microsoft SQL Server and PostgreSQL.

While in development, there are different ways of setting up a database server for us to work with. They can be run locally, which requires installing the required software on our own computers and will use some of their resources while running the database server. They can also be run on the cloud, which simplifies the process by a large margin and spares our local hardware resources from the burden of having to run the database, but it might add extra costs or come with some limitations, for example limiting the number of rows allowed, as it is the case with Heroku (Heroku Dev Center 2021). One last way of running our database server, which is becoming more popular nowadays, is using a container, for example with Docker. Containers, which can be run locally or on the cloud, offer a great advantage when setting up the database server which is making the process much simpler and faster as well as making it easy to reset the database, both very useful while in development. Containers also allow us to control our application environment and thus preventing errors that might arise, which is why they are also very popular for deploying applications.

2.1.2 Programming language: Why Typescript?

The Web Applications were originally written in JavaScript -or JS, to put it shortly-, which is the language we have used the most during our Software Engineering courses in Haaga-Helia. JS is a good choice since it is a very popular language. It has often come on the top spot of most popular development technologies, as can be seen from Stack Overflow's Yearly Developer Survey which shown to be used by virtually 65% of all respondents and close to 70% when taking into account professional developers in the latest publication (Stack Overflow 2021). Being supported by most browsers, it is often the default choice in front-end development for creating interactive websites, while adding the fact that it can also be used in the back end with a runtime like Node.JS means that we can use one language for both. This is a great advantage for a Full Stack developer as it greatly simplifies the work and enables them to focus more on the quality of the code and the features of the program than in the quirks of each language and syntax differences.

As the projects progressed and I chose them to be part of this portfolio, the idea of migrating to TypeScript came to mind. Why embark on such a task? There are many reasons which I will go into below.

The first reason is that TypeScript (or TS) isn't a programming language of its own, but actually a superset of JavaScript, which means that all valid JavaScript code is also valid TypeScript code, but TS adds an additional layer on top, its *type system* (TypeScript Docs 2022). What this means is that if someone has experience developing software with JavaScript, migrating to TypeScript is a much simpler task than switching to another programming language and all previous knowledge and experience remain useful.

This, however, is not the only reason. Migrating to TS has also a few advantages over JS. While JS is loosely typed and dynamic, meaning variables are not directly associated with any particular type and can be (re-)assigned values of all types (MDN Web Docs 2022), TS features static type-checking. This can help identify bugs early in the development process, even before running the application which can also save us time -and, sometimes, frustration- and becomes more significant as the project's scope and complexity grow. Type annotations can also function as code-level documentation (Torppa, Peuraniemi & Rapo 2022) and integrate with Integrated Developer Environments (IDEs) to highlight errors or troubling syntax and suggest fixes.

The last reason is that, while not as popular as JavaScript, TypeScript is also a rather popular language and in good demand for working life. While it was mentioned that JavaScript came in first as the most used programming language in Stack Overflow's annual survey, TypeScript, despite being newer and not as popular, still manages to come out in the top five among professional developers, as seen next in figure 1.

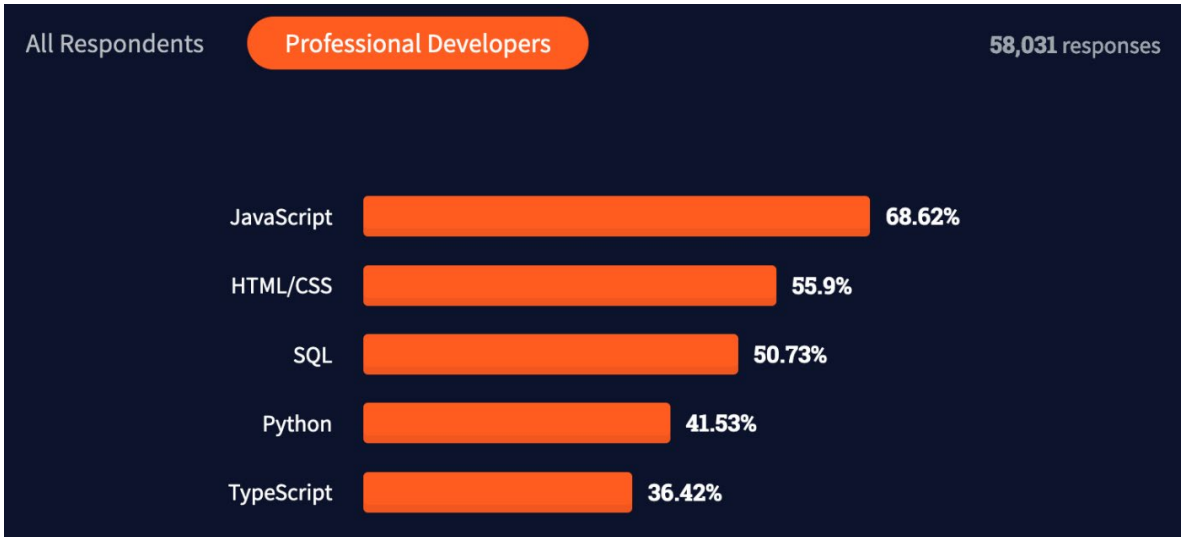

Figure 1. Most popular technologies. Programming, scripting and markup languages (Stack Overflow 2021)

But this is not all; figures 2 and 3 show that TypeScript comes in second and third for most loved and most wanted programming language respectively and, in both cases, it comes ahead of JavaScript.

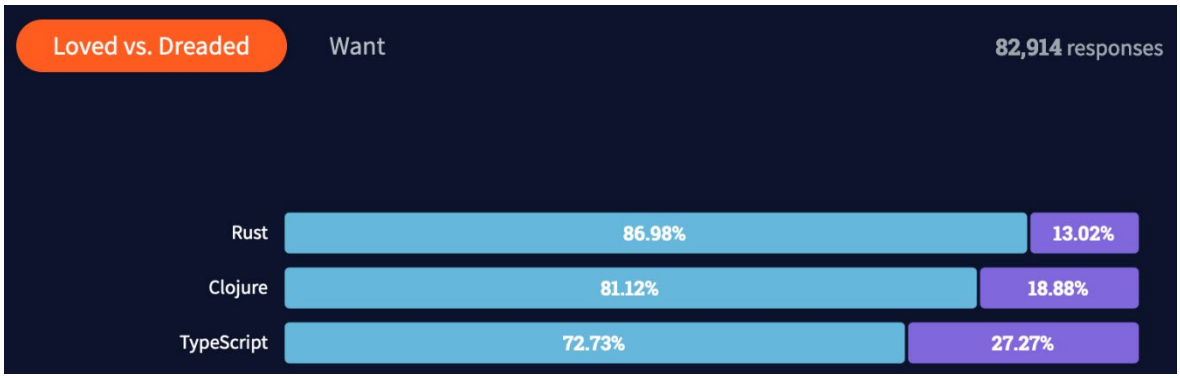

Figure 2. Most loved programming languages (Stack Overflow 2021)

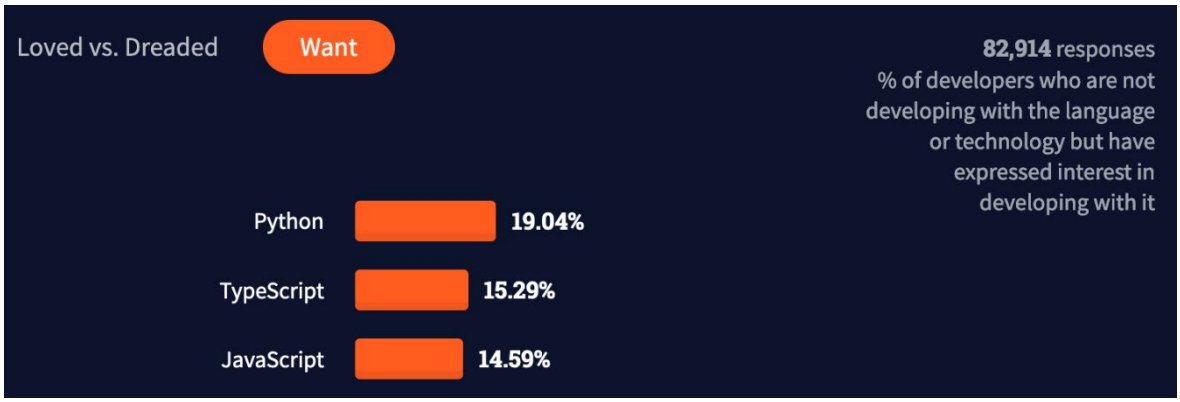

Figure 3. Most wanted programming languages (Stack Overflow 2021)

The “Loved vs. Dreaded” chart in figure 2 shows how satisfied -or unsatisfied- developers are with a certain language, while the “most wanted” chart in figure 3 ranks languages by how much developers want to work with them, if they are not currently doing so (Stack Overflow 2021).

For all these reasons, I will be refining the Full Stack products by migrating them to Type-Script so they can increase the appeal of this portfolio.

2.1.3 Back-End Applications and REST APIs

A back-end server can provide data to the clients/front-end applications. This is usually done with a REST API.

API stands for Application Program Interface and “is a set of definitions and protocols for building and integrating application software (…) sometimes referred as a contract be-

tween an information provider and an information user" (RedHat 2020). They work receiving requests from the clients and sending back responses using HTTP. Resources can be accessed via endpoints, which can be either public or private, in the latter case the clients needing authorization in order to access them.
REST, or representational state transfer, was introduced by Roy Fielding in his dissertation. It is described as an "architectural style" (Fielding 2000, 76) based on constraints which APIs should adhere to in order to be considered RESTful. REST emphasizes the separation of concerns between the client or business logic and the logical data storage, which allows more independence for both the client and the server programs, and constraints the communication between them to be stateless, meaning that the server should not store any session state and any requests should contain all the information necessary in order to be properly handled, simplifying operations significantly for the back-end server (Fielding 2000, 78-79). The information provided is defined as a resource, and accessible through a resource identifier. The method used in the HTTP request will determine which action is taken on the resource; the most common are GET -which is the only one which does not change the resource-, POST, PUT -which can be used to create or update a resource- and DELETE.

Because the process of writing a back-end server often requires doing the same steps, as there are many features such as routing, connecting to a database and/or authentication which are found in most implementations, it is common to use a framework for the task. Some of the most popular frameworks used for creating back-end servers are Spring using Java, express.js using JavaScript running on Node.js, Microsoft's ASP .NET Core using C#, and Laravel with PHP. Most likely, the framework will be chosen based on the used language.

To run JavaScript outside the browser requires a runtime, which is where Node.js comes in. "Node.js runs the V8 JavaScript engine, the core of Google Chrome, outside of the browser." (Node.js 2022). This will mean we don't have access to browser-exclusive features such as the DOM -Document Object Model- and some web APIs, but in exchange Node offers features, for example accessing the file system, which aren't available in the browser. Node's functionality can easily be extended thanks to the large number of libraries available for it, which are easily managed with npm -node package manager-.

2.1.4 Front-End Applications

In web development, front end refers to the front-facing side of a website, that is the user interface which will be interacted with. An important part of front-end development is focusing on the user experience. A good application should work well, be easy to use and

also to learn, meaning interacting with it should be intuitive. It should also be free of bugs and errors, but debugging applications of this kind can be particularly challenging, as real users are hard to predict and the number of variables which can “go wrong” increases exponentially.
Traditional websites were built using HTML and styled with CSS, with JavaScript added to enhance interactivity. As sites became more complex and evolved into what we call web applications featuring more intricate designs and interactions from the user, AJAX, or asynchronous JavaScript and XML, which allows for the loading of data without the need to reload the whole page was added. This data can be fetched from an API, like the provided in the back end previously explained. Frameworks and libraries also appeared to assist during the development process. Some of the most known ones are the older but still popular jQuery, Angular by Google, React from Facebook, and the newcomer Vue.js. These are all open-source.

React is the library used in the front-end applications part of this portfolio for building the user interfaces. It is not a full-fledged framework, which means not all features are present out-of-the-box. For example, routing requires an external library. That said, some of its biggest advantages are:

- The amount of documentation, tutorials, and support available, thanks to its sheer popularity.
- The ability to add components and libraries or create our own to be reused, saving the developer(s) valuable work time. The number of third-party libraries is also quite large.
- A data model built to prevent errors and unnecessary re-renders, based on data flowing downwards, as properties from parent to children component, and states managed inside a component, which can be used as a “source of truth” to keep track of data.
- A virtual DOM, which can be used to improve performance when applying changes to the actual DOM.

As the official website says, React is declarative. Declarative programming is a paradigm where the code written describes what should happen (or, when building UIs, what should be shown) instead of how to achieve those results. With React, we describe the final output, and the library will handle the updating, rendering, and maintaining UI and model synchronized. The render() function returns a tree of React elements at a given point in time, like a snapshot. When an update takes place (of a state or props), React will rebuild the tree if the root element is a different type; if React DOM elements are of the same type, it will update any changed attributes (and only those); it will also mutate children elements if it finds any differences between those after performing a comparison, recommending a unique “key” prop for each to improve efficiency when performing this process (React 2022). We will see, in the next chapter, that SwiftUI also adopts this paradigm, probably inspired by React and its popularity.

Regarding the architecture of front-end applications, a modern style is seen in Single Page Applications, or SPAs. These types of applications load the full page at start and then dynamically update content without loading a new page. This separates them even more from traditional websites, closer to what would be a native application. It improves performance and responsiveness, enhancing the user experience and better supporting client-side behaviours (Smith 2021, 8). The frameworks and libraries previously mentioned are ideal to work with SPAs.

One last topic to cover in this sub-section is regarding deployment of front-end applications. While some cloud services automate the building and deployment process, it can also be done by the developer when a greater control over the setup is desired. When using React, Vue or Angular, the first step is to create a production build for deployment. This is done from the command line and the whole procedure can be found from their documentation pages. Once the build is ready, it needs to be served by a web server, either locally on the computer or on a remote infrastructure. It can also be served by the same web server as the backend. If they are on different web servers, CORS errors might occur in the browser, which can be solved by proxying from the front-end application to the back-end API.

2.1.5 Additional Technologies I: Cloud Services

Cloud services can be a very useful tool both during the development process and when deploying to production.

Heroku has been used occasionally during development. It is a Platform-as-a-Service, or PaaS provider, meaning the user doesn't need to bother with the underlying infrastructure. They provide app containers called "Dynos" to deploy web applications and also cloud databases, including Postgres for SQL and a NoSQL option. The free plan allows all these to be used, although with some limitations. The only major limit for app Dynos on the free tier is the "sleep on inactivity" mode (after 30 minutes) which will make initial loads take slightly longer after wake-up, but for development, education, or personal projects, this is not a big issue. For Heroku Postgres, the free version is limited to 10000 rows and 1GB of storage; whichever exceeds will results in limitations being applied, such as INSERT commands being disallowed. Other limits should not negatively affect the normal functioning of the app during development, as they are mostly limiting the service for production use. The full list of limitations can be found from Heroku Dev Center (2022). By linking a GitHub account, deployments can be carried out directly from a chosen repository.

Overall, the service focuses on simplicity and can aid during the development process and deployment even for a personal project. Heroku uses Amazon Web Services, or AWS, for the underlying infrastructure which can guarantee good performance, uptime, and security, including compliance with industry standards (Heroku 2022).

The second platform used during the work done for this portfolio is Microsoft's Azure. This one is much bigger with plenty of services on offer, which range from easily accessible to those targeted to highly skilled professionals. There are some free services offered, and students are also eligible for some free credit, an offer I made use of while working on these products. Azure's App Service is a similar offering to Heroku and can be used for deploying web applications. It also supports deploying directly from a GitHub repository (as well as other platforms), plus containers and it even integrates with many IDEs. App Service can run on a Windows or Linux server (Microsoft 2022). Azure has a large number of databases available, including SQL Server, MariaDB and PostgreSQL. On the lowest tier, the database has a 5GB limit, but no row limit. The app service features a free plan and Azure SQL Database does too, but the PostgreSQL one does not. For students, Microsoft offers 100$ credit when using an email from an educational institution. Azure also offers virtual machines, where users are free to configure and setup their deployments as they see best, which can have some advantages but also requires much more work.

Both services offer good documentation of their features and guides to assist with using their provided tools.

2.1.6 Additional Technologies II: Version Control

When working in software projects, especially as part of a team, a version control system is a very important tool to have. From the book Pro Git (Chacon & Straub 2014, 8), "It records changes to a file or set of files over time so you can recall specific versions later". This helps maintain a smooth development process, keeping track of changes and being able to revert them if needed, as well as documenting those changes and the process itself. Doing this in a structured way means it is less likely for something to go wrong, as opposed to, for example, simply backing up files. Git was created in 2005, originally by Linus Torvalds, with focus on being distributed -for lowering the risk of failure-, fast and simple to use, supporting large projects and non-linear development in the form of branches (Chacon & Straub 2014, 12). Git works locally by cloning the repository, along with all its history. This makes further actions quick to perform, as we don't work with the remote source. As well, we can make changes and commit locally, and push them to the remote

server when suitable. One more way in which using version control eases up the development process is that it allows developers to try to implement new features or fixes and revert to the last good “checkpoint” if the results are not satisfactory. Git is quite popular and there is plenty of documentation available.

GitHub is a platform where users can host their code in repositories. Users can also contribute to others’ projects, either directly -if access has been given- or by forking their repositories. In the second case, changes can be also brought back by submitting a pull request (if it’s approved). GitHub can be used from the command line using the git client or GitHub’s own CLI client, or using the GUI version, called GitHub Desktop. Other services have similar features. GitHub also supports DevOps and automation with GitHub Actions, where we can set up workflows for automated testing, building, and deploying of our projects at certain points (for example, when there is an update to the main branch), and it has the ability to keep track of issues and wanted features.

One last thing to keep in mind regarding the use of a version control hosting service for our projects is that we can add a README to it. This file is very important because it is shown on the repository along with our code, and it is the first thing anyone will see when they open it. Whether that person is someone who we are showing our project to, such as someone who is taking a look to the products included in this portfolio, or a new member of the team which is joining to work along with it, the readme is from where they will obtain background information about the project (e.g. what is its reason to exist, what kind of project is it, which technologies does it use, etc.) as well as important steps to be followed when setting it up and using it. Finnish company Solita was kind enough to provide some guidelines for applicants attempting their pre-assignments with tips and useful patterns to follow, and from their article we can find good rules to try to keep when attempting to write a good README file. It should include:

- As mentioned, a description of the project is important to have. The readme can include technologies used as well as the reason for their choice, if there is one.
- What do we need to have or to do in order to be able to run the project? This should include any additional software or dependencies which must be installed beforehand.
- Are there any special configurations which need to be set? In such case, it’s important to be clear and specific as to what needs to be changed and where.
- For Full-Stack applications which are completely under one repository, it might be necessary to include instructions for both the backend and the frontend, in some cases as far as giving each their own section.
- If the project features tests, more information can be given on these as well as how to run them.
- Missing or planned features. This is particularly useful for an ongoing project.

(Vainio & Valleala 2021)

2.1.7 Code structure and maintainability

One last subject to cover in this theoretical framework are the standards which the code within these applications should aspire to meet, as this is a very important topic when dealing with professional projects.

Good code should be easy to read, understand and maintain. This is due to the fact that very often projects are not handled by one person alone. When working in a team, it is of the upmost importance that any member of the team understands how the application work to its innermost details and while certain decisions can be up to personal preference, some patters can be kept to ensure the code written is clear. There is a lot of information, both online and in printed form, about best practices of software development so in order to keep this chapter from growing too large or overreaching the scope of this thesis, I will list some guidelines I have used. Some of these I have come across doing my own research and some have been pointed out by colleagues (with proper references).

For the sake of maintainability and further development of an application, there are some design principles which can be followed and are well explained in the Microsoft-provided eBook “Architecting Modern Web Applications with ASP.NET Core and Microsoft Azure” by Steve Smith (2022, 10-15). Here is a summary:

- Separation of concerns: separating the code based on the kind of work it performs; in other words, avoid mixing different concerns in the same class, function, or project. A good example from web application design is the three-layer approach, consisting of the data access layer, the business logic, and the user interface.
- Single responsibility principle: In object-oriented programming, states that objects/classes should have only one responsibility and one reason to change. When applied to services, the same applies; every service should have a single responsibility, with the ultimate example being the microservices type of architecture.
- Don’t repeat yourself: repeating logic can lead to errors and make future changes more complex, so it is best avoided. Some ways not to repeat ourselves when writing code are, for example, refactoring repetitive code into functions, group logic into encompassing classes and even into projects (for example, libraries or modules).
- Makes classes depend on abstractions -interfaces- and not implementation details. This is called dependency inversion and should go together with the “Explicit dependencies principle”, which states that methods and classes should explicitly state their requirements in order to function (which also makes the code better at self-documenting).
- Make the business logic infrastructure/database independent.

There are also more practical rules we can follow. A good start, as pointed out by Robert C. Martin in his book *Clean Code*, is with names. Variables should have descriptive names, showing what the values which they store represent. Functions should follow the same rule, so that the name itself describes what they do. Function names usually contain

a verb in imperative form for what the function “should do”. Since functions are being mentioned, it’s also good to always remember to keep them as small as possible -again, to make them easier to understand- and ideally doing one thing. (Martin 2009, chapters 2-3)

Comments should be kept to a minimum and are mostly used in cases where the code isn’t completely clear, exceptions have been made, and similar. Comments are also useful in configuration files, used to explain why certain default settings might have been overridden.

When working as part of a team, it is also very important to take care of actions when using version/source control. First and foremost, being clear when making commits; there should be a description regarding what was changed. Common cases would be when something has been added, fixed or removed. With verbs, the convention here is to also use the imperative (Chacon & Straub 2014, 130-131). Descriptions shouldn’t be too long in order to remain easy to read, and more information can be added to the body -what the problem was, how has it been improved/fixed, etc- (git 2022). Keeping the commit messages short also means they should ideally not encompass too many changes. For pull requests, the description can say what is being fixed, and if some project management software is being used it can contain a link (or the name in the title) of the issue being worked on, for example a Jira ticket or GitHub issue.

Writing clear and maintainable code is a very valuable skill. It means our programs will be easier to work with and to fix, which is helping not just ourselves, but also others who might need to do it either now along with us or in the future.

2.2 Empirical part

The empirical part of this thesis begins with the first two products of the portfolio being shown, what I consider my best examples of Full-Stack web applications which have been done entirely on my own. This part aims to show my skills and understanding in the whole process of developing applications of this kind and scope. I will begin describing the requirements of each task (in other words, what need are the applications supposed to address), the application structure along with technology choices, the development journey coupled with examples of the finished products and my overall view of the end result. Further reflecting and considerations will take place on the last chapter of the thesis.

Both tasks were done as part of pre-assignments or challenges for job applications with well-known Finnish software houses. The companies themselves were not involved in the project any more than giving the basic guidelines and requisites so they were not done “in

cooperation". This thesis will include links to the GitHub repositories with the source code for each application. These projects have been some of the most demanding which I have done, especially -but not only- because of time constraints and they encompass many different aspects of software development and technologies which makes them a better example of my skills learnt. The projects also introduced some new technologies which I had to research on my own. This presented a great opportunity for me, and I will describe these in detail as well.

As part of the thesis' work, I continued working on them to further improve them based on my own learnings during my research and thanks to feedback which I have received since submitting them. This has been very valuable as it enables me to continue learning and improving. If there are any aspects which exceed the scope intended for the work of this thesis, I will make a mention of them, including a more detailed description and why they are of significance.

2.2.1 Farms Application

The first product shown has been created as a pre-interview exercise. The repository with the source code is hosted here: https://github.com/ernven/dev-academy-fullstack-project

The premise was that there was a need for a Full-Stack web application which displays data from different farms. The data would be coming from CSV (comma-separated value) files which would need to be parsed and entered into the system. There were some validation rules, which needed to be taken into account regarding the data being entered, and a list of optional features which would make good extras to show the applicant's skill, although it was mentioned that they were only suggestions and not requirements. The solution delivered had to fulfil the needs regarding the application's functionality, contain good code, any features which were implemented should be completed and working, be presented by a good readme, and containing tests.

The application is composed of a PostgreSQL database, a backend running on Node.js and a front-end web client built with React.

For the database, I have used two different set ups. During development, I used an Azure-hosted Database for PostgreSQL server. For deploying to production, since I deployed the app to the Heroku platform, I decided to set up a Heroku Postgres instance as it's easier to work with when both are under the same service. Heroku apps do not have static ip addresses on the free plan (or even somewhat limited ranges) which means I would have had to open the firewall on the Azure database to the whole internet, or at least all AWS

ranges I could find. The application code contains SQL statements which can be used to create the necessary tables for the app to work. Executing these was done using pgAdmin, which was also used to query data and perform maintenance on the schemas. The database structure consists of two tables and can be seen as following in Figure 4. These were designed based on the provided data and validation rules.

`farms`

Column	Definition
id	Type UUID, NOT NULL, auto generated. Primary Key.
farm_name	Type VARCHAR(50), NOT NULL.

`entries`

Column	Definition
entry_id	Type bigint, NOT NULL, auto generated. Primary Key.
farm_id	Type UUID, NOT NULL. Foreign Key, referencing farms.id.
date	Type TIMESTAMP, NOT NULL.
entry_type	Type VARCHAR(11), NOT NULL. Must be either 'pH', 'rainFall' or 'temperature'.
read_value	Type NUMERIC(5,2), NOT NULL. Must be between -50.00 and 500.00, per project guidelines.

Figure 4. Farms app database structure

The Azure-hosted database worked rather well. The one on Heroku was also performing decently, but because of limits to the free tier, when more than 10000 rows would be added, which was not enough to store all the sample data provided, it would give a warning in the admin panel and limit further INSERTs after 7 days had passed. Luckily, I was able to show the application working before this came into effect, but it does restrict the kind of apps we can build and show with the free tier.

The back end was built using the express framework. I was already familiar with it, from working with express during our Software Project course in 2020, which made it a good choice -again, considering the limited time I had to work on the project-. Some of the things I like about express are that it's very simple and clean to set up, that it's a Node.js framework which means it can be used with JavaScript (and TypeScript) and its great support of middlewares to easily add functionalities to our API. To use the back end to perform actions on the database (queries, inserts, etc.), the API provides many endpoints. These are documented in the project's README. In short, we can obtain a list of farms in the system as well as all the data using GET requests. Sending a POST request to the same endpoints allows us to submit new farms or data entries, which was a requirement for the task. Data fetched can also be filtered using query parameters, for example by sensor type or farm name, or period, using path parameters for year and month. There

are also endpoints for getting calculated data, such as averages and extreme values, as well as one which serves data in a format which is best suited to build charts using the chosen front-end library (recharts). For the back end to work, some environment variables must be set first. They are also detailed in the project repository's README.

One thing I had to learn while developing this application, based on the project requirements, was how to upload data from a CSV file, including receiving the file in the back end, parsing, and saving the data to the database. This took me a bit longer time, but it was a great experience and very engaging. I believe it will come in useful in the future as CSV is a common format for data distribution. The operation required that the body of the request is parsed based on its multipart format and then the CSV data itself is also parsed. For this, two libraries are used, busboy and csv-parse.

The structure of the back end is standard for an express app: A *config* folder contains configuration files. The index.js file is the entry point for the back-end server. The *routes* folder contains the different routes used by express, while the *controllers* folder contains the controllers which make up the intermediate layer between the routes and the infrastructure (in some architectures, this intermediate layer is called the Service layer). Because the project uses the Knex library for accessing the database and building queries, it's not necessary to have a separate infrastructure/database access layer. Other helper modules and libraries are inside the *utils* folder.

The front-end application features different components to add new data into the system and display existing data in different formats, such as tabular, graph and using a dashboard. The insertion of data is done in the admin portion of the app, which would require authorization in a commercial product. The rest could be available to all users, but specifications could vary depending on the client's needs (this was not specified in the guidelines). The app uses a few libraries to facilitate development. AG-Grid is used for displaying data in a table and it is very well suited for large datasets, thanks to implementing virtualization of its structures, which means rows are rendered as the user scrolls instead of everything at once, at all times. I have found out about this library during development of this project for this exact reason. The first implementation used react-table, which I chose as I had already used it in the past and am quite experienced with it, but did not perform well enough in this scenario. The library ReCharts was used for displaying graphs and charts. I had not work with it before, but it is one of the most popular charting libraries for React. It does the job fine but is very particular about the way the data needs to be formatted; this made me build an endpoint on the back end solely for providing data for this component. As for the User Interface, it is built using Material-UI, which is also a very popular

library and it's based on Google's Material Design guidelines. We have used it in many courses in Haaga-Helia.

In order to refine my application, I decided to migrate the source from JavaScript to TypeScript. This way I could also learn more about TS and how it works. I did this after working with the next application, so I was already somewhat familiar with the process, but it was in any way good practice. Some obstacles which I faced were regarding library compatibility and support for TypeScript, the latter was particularly troublesome with Material-UI, as finding and declaring types for all the custom components, their props and themes becomes like navigating a maze. Part of the code had to be cleaned as well, especially in the back end. Some of the controller functions ran pretty long, and their names were not the clearest, so the code was made a bit more readable and easier to maintain.

The main screen of the application, seen next in Figure 5, shows the header, a welcome message, and a navigating menu on the left side. The menu shows by default icons only, but clicking on the "hamburger" button expands the drawer menu so the items contain text as well as the icons (as shown in the picture), which might be useful for users new to the app.

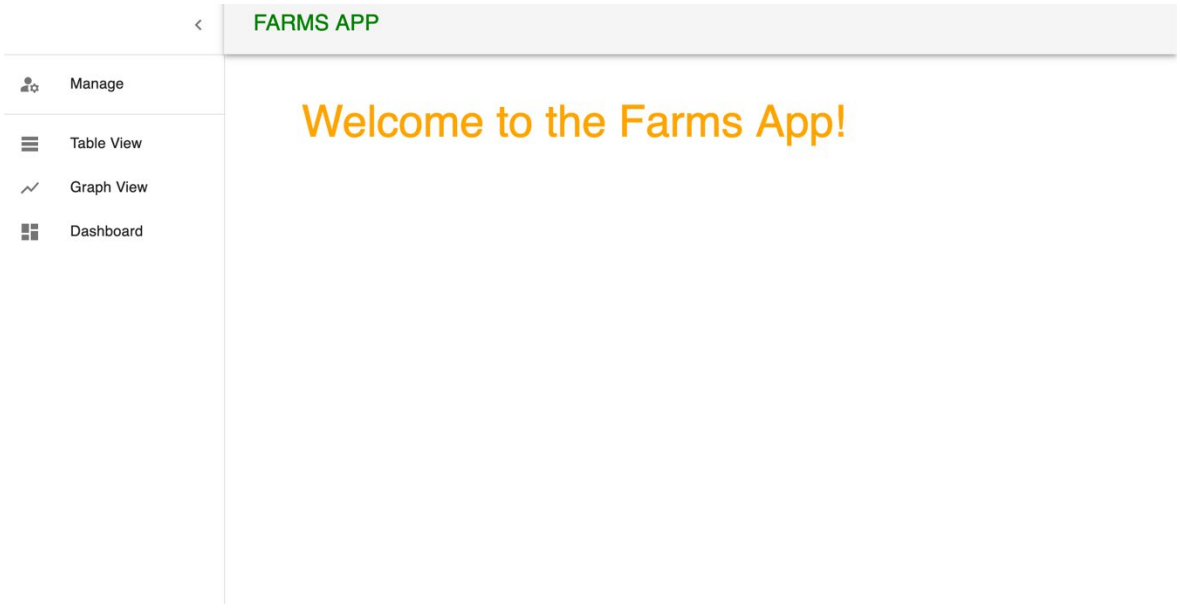

Figure 5. Main screen of Farms Application

As we see, the options available are the admin portal -under the title "Manage"-, a Table View of the data available, graph views and the data dashboard. The admin management portal is shown in Figure 6 and is rather straightforward. There is a text input field to create a new farm entry in the system and a file input dialog to upload data files. Both fields have validation implemented handling the more usual cases for errors, for example dupli-

cated or empty farm names, and file formats which are not supported (per project requirements, only .csv files are handled). Also, the file upload dialog shows the size of the file selected for submission.

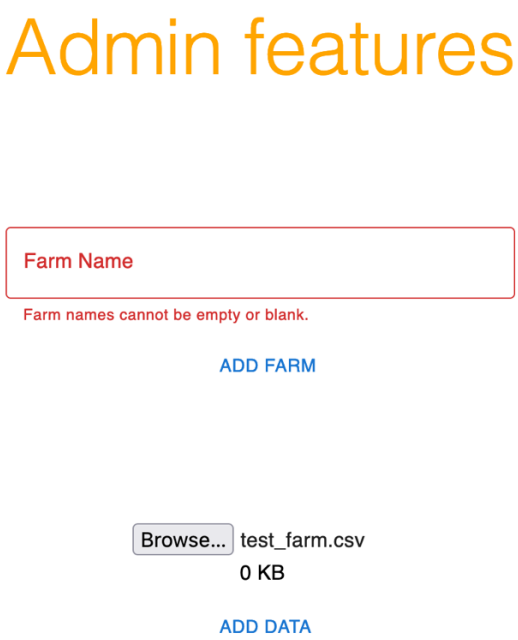

Figure 6. Management screen

Figure 7 shows the Table displaying all the data currently in the system. At the moment, even with a large dataset, the component handles the rendering quite well (thanks to row virtualization), but if the amount of data available was to grow much larger, some more advanced techniques could be implemented to ensure good performance, for example lazy loading of the data -in other words, load data as needed- or showing individual filtered results at a time, for example only a certain farm, reading type or period of time.

Farm Name	Date	Type	Value
		ph	
Noora's farm	20/02/2019, 21:41:47	pH	6.46
Noora's farm	21/02/2019, 04:54:11	pH	6.40
Friman Metsola collective	21/02/2019, 10:41:43	pH	6.05
PartialTech Research Farm	21/02/2019, 13:38:50	pH	5.86
Organic Ossi's Impact That Lasts plantase	21/02/2019, 20:41:54	pH	7.21
Noora's farm	22/02/2019, 06:21:46	pH	6.18
PartialTech Research Farm	22/02/2019, 07:18:56	pH	6.22
Friman Metsola collective	22/02/2019, 16:58:56	pH	6.08
Organic Ossi's Impact That Lasts plantase	22/02/2019, 20:10:34	pH	6.80

Figure 7. Data table screen

Figure 8 shows an example chart. This was created simply for demonstration purposes since the possibilities for data display are endless and it would depend on what would bring value to the users/clients. In the example chosen, the graph shows average daily values of a certain reading, showing temperatures in this figure, for all the farms together, each represented by its own line. We could also show other types of date, aggregates, etc.

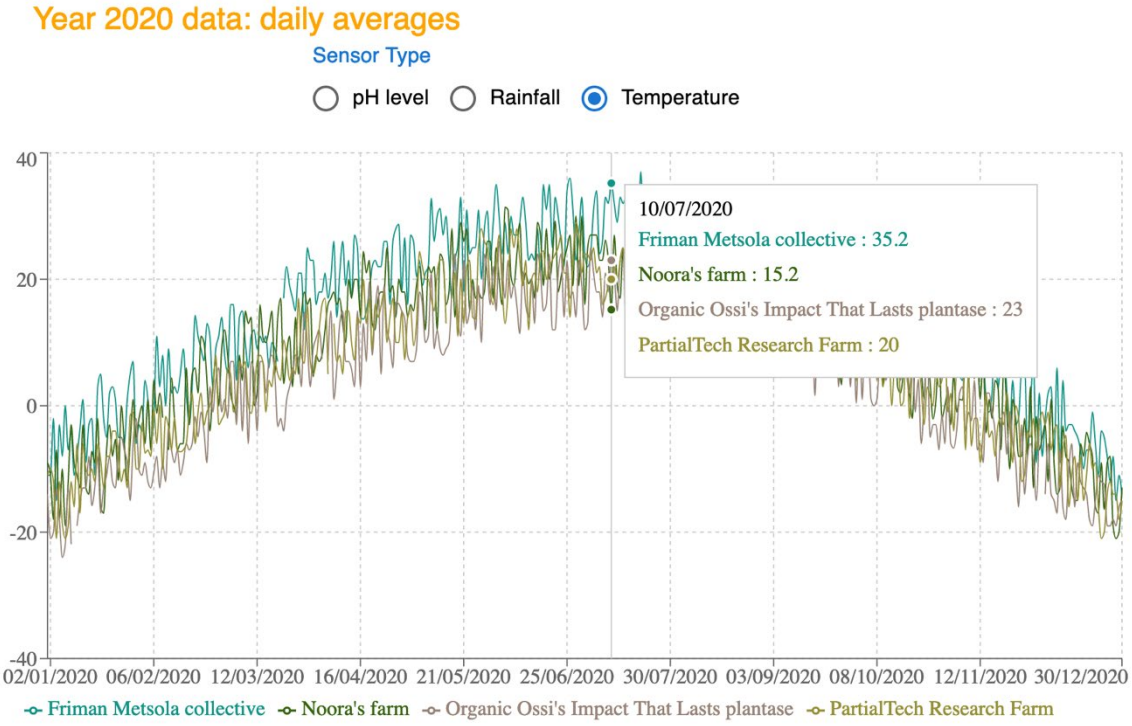

Figure 8. Chart example

Lastly, the dashboard in figure 9 shows data for a single farm.

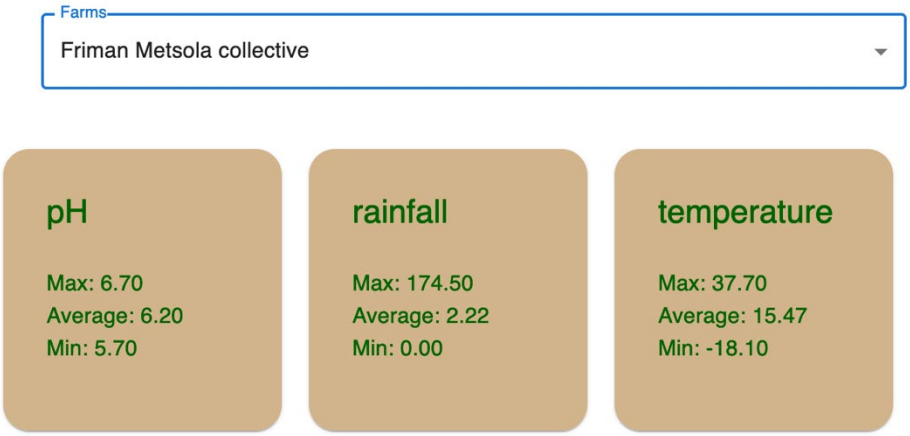

Figure 9. Data dashboard

The data chosen are calculated values, in this case the maximum, average and minimum values for each sensor.

Because the purpose of this project, as well as the next one, was and is to show my skills developing a Full-Stack Web Application, the focus is with the technologies, the code, and the functionality. This, as well as the time constraints faced, is why I chose to build a clean, functional user interface (using Material-UI) without overproducing the visual aspect in a way it would have been detrimental to the quality of the system and its functionality.

2.2.2 Rock, Paper, Scissors Results Application

This next product was also done as part of a pre-assignment. The project repository can be found here: https://github.com/ernven/reaktor-2022-fullstack-version

In this project, it was asked for a web application which would display results of “Rock, Paper, Scissors” matches. The full history of results came from a so-called “bad api” (this term was used by the project itself) which had only one endpoint. This endpoint had the whole history, served in different pages. To navigate pages, each result page would contain a cursor to the next, until the last page is reached. The API was “bad” as it failed to provide the data in a more efficient manner. This, coupled with the fact that the historical data had an enormous number of entries, meant that the application would spend a lot of time fetching, affecting performance. One of the requirements for the project was that the app should work fast and display data to the user without delay, so one task was to come up with a way to solve this problem. Besides historical data, updates would come in real time using WebSockets. The docs at Mozilla developer network describe the WebSocket API as enabling “sending messages to a server and receiving event-driven responses without having to poll the server for a reply” (Mozilla Contributors 2022). This means that by having the connection established between the client and the server, the server can send updates when they are available, without the client having to request them (either at regular intervals or when needed). Although communication using WebSockets can be bi-directional, in this case it was only used to push updates from the server to the client, in a way like what can be accomplished using Server Sent Events. As part of the requirements, the live games must be always visible, and historical results must contain all games by player and include some aggregate data, such as win ratio, number of matches played and most played hand (Reaktor 2022). Similar to the previous assignment, special attention was paid not just to look and features, but correct functioning of the application, writing clean and maintainable code, technologies used and performance of the application.

Since the project already provided an API, albeit a “bad” one, I first attempted to create a solution comprised only by a front-end application. This app was also created using Re-

act, and I tried to architect it in a way to make the execution as quick and smooth as possible, and to give a good user experience by working around the limitations of the API. Some ideas I had was to load the data progressively in the background, making sure that it did not affect the loading and correct functioning of the app and other features. This worked in the beginning, as I could display the UI and live games while the historical data was loading in the background, but if the user would want to see those results right away, they would not be complete. To solve this, I thought of the possibility to cache the historical data as well as the cursor in LocalStorage, so that the app could resume loading from the last found page and not have to start over from the beginning. The problem I ran into then was that the historical data was so massive that the size limit which can be taken by LocalStorage (between 5 and 10 megabytes, depending on the browser) was not enough to hold the data. This left me no choice but to implement my own backend to bridge between the “bad api” provided and the frontend client. I again opted for familiar technologies and developed the back-end server with Node.js and express and used the same Azure-hosted database server for storing the data. Using technologies and frameworks which one is used to working with is a good decision for projects like this one, where time is limited, and it is best not to negatively affect the quality of the end product by improvising or learning something completely new.

The structure of the final product is traditional of a PERN stack application and similar to the previous one. The repository has instructions for setup and building the app, which settings need to be defined and the “scripts” folder in the backend contains the SQL required to build the correct database structure. For portability and security reasons, environment variables are used for storing configuration settings. In this case, having two APIs serving data, special care must be taking setting these. The WebSockets URL must be set in both the backend and frontend, as they both receive results in real time; the backend saves this data into the database and the frontend displays it to the user.

The backend sends requests to the provided API for the historical game result data and saves it to the database. This is done using the library Axios, as fetch is not available in Node.js (in other words, outside the browser). Axios is straightforward to use, and even automatically parses the response body into JSON which allows that step to be skipped in the application code. The data is formatted in a way it makes it easier to be retrieved in a more efficient manner, based on the requirements of the project. The database contains two tables: players, containing the names of all players, and games, with game details such as players involved, and hands played. When run, the backend server will start fetching data and establish as well the WebSocket connection to receive real time game updates. These are also added into the database, to stay up to date with the latest data. Data is served to the client with two endpoints, also for games and players. The endpoint

to retrieve historical game data is where the biggest gain is achieved, as we can obtain all data at once and filter it, in this case by player, meaning the frontend application can request only the data needed.

The frontend is also a standard React App. The user interface is simple and presents most information at a glance; the live games are shown as soon as the application launches and historical statistics for players can be searched right from the main screen, as can be seen below, in Figure 10.

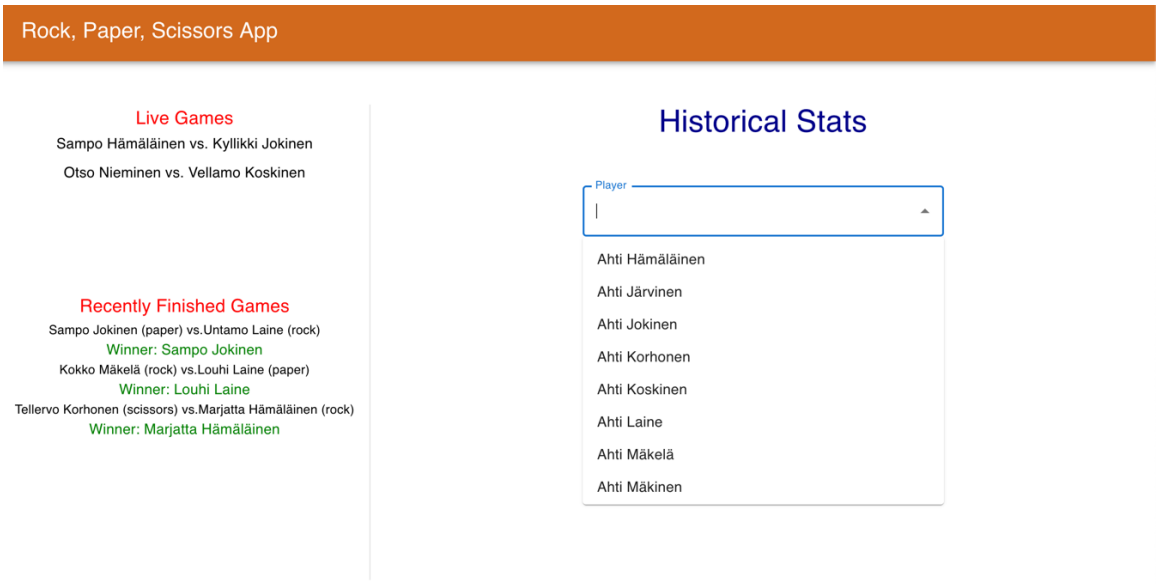

Figure 10. Main screen of Rock, Paper, Scissors App

The interface is done using Material-UI, with the aim again of presenting a clean, modern look without overcomplicating the task of handling the visual aspect. The next couple of Figures show the components displaying player data.

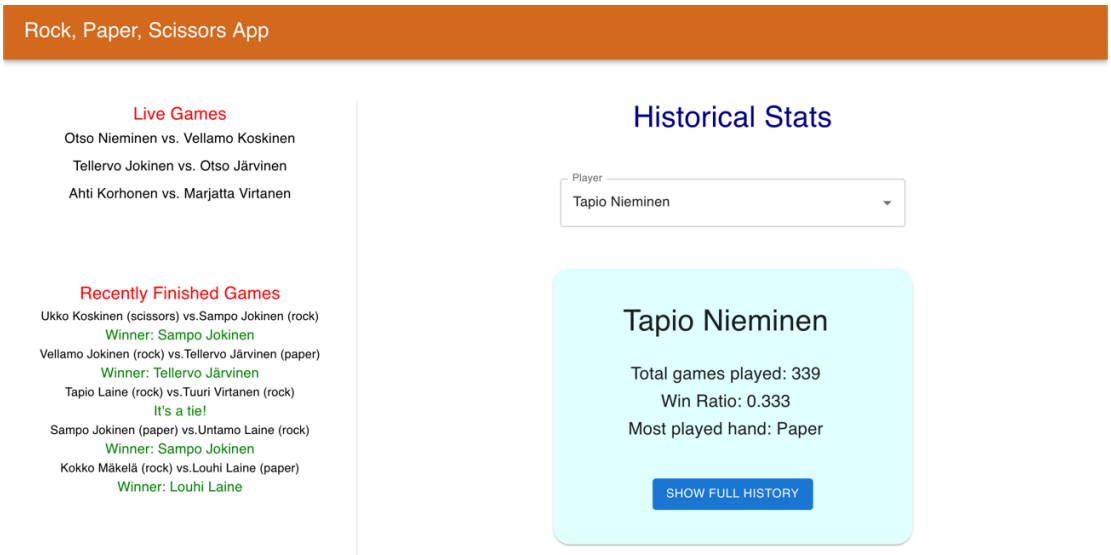

Figure 11. Screen with player stats card

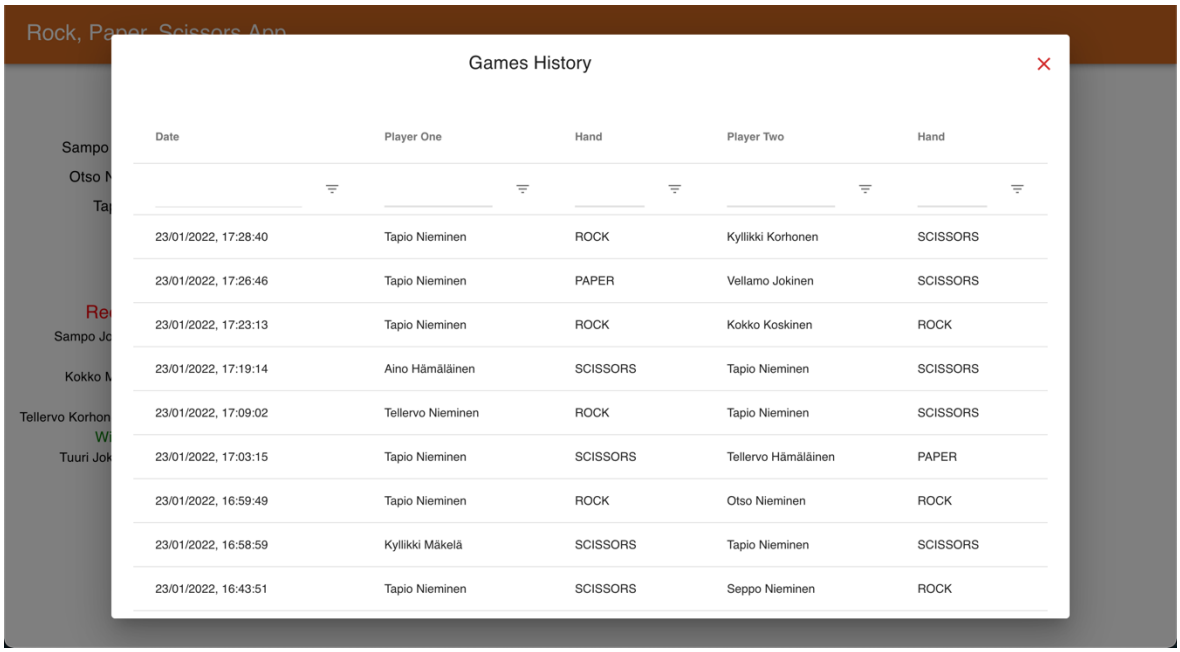

Figure 12. Games history for a chosen player

In Figure 11, the card displayed shows calculated data for the chosen player, while Figure 12 shows the table view displaying the full historical data for said player. The table is built with Ag-Grid to ensure good performance even with large datasets.

The front end gave me the opportunity to learn two technologies while developing this product, which I am happy for and will describe next. First, because of how the data is provided in real time, I had to learn how to work with WebSockets (this was also done in the backend but was first implemented in the front-facing side). Although I had experience building an application which should handle updates from a back-end server, experienced I acquired working on the Multidisciplinary Software Project -which also was basis for the iOS app also part of this portfolio-, I had only implemented such using Server Sent Events. WebSockets are slightly more complex, but with one-directional communication the difference is not significant. I managed to learn how to work with it and implement the real time feature, which receives and parses the data to be displayed, into the RealTime component. With some refactoring, the communications could be extracted into its own module to improve testability and maintainability, as well as separating data and UI logic. The second feature, on topic with the data discussion, is the use of reducers to manage state. Reducers have become popular thanks to the library Redux for handling more complex state and data structures within an application, but nowadays they can also be implemented using Hooks, for applications where complexity and scope is not big enough to warrant an external library. In this app, it has been done with the useReducer Hook. This takes two parameters, the defined reducer function and an initial state, and returns the current state and a dispatch action. The reducer function takes the state and an action; the action contains a type, based on which is how the data will be handled, and a payload,

which is the data passed to be handled. The concepts are somewhat hard to visualize just by explanations so, based on this application, an example would be that action types can be whether a game has started or ended, and the data are the details of the game; depending on whether the game just started or ended, we will handle that data differently (e.g., saving the results). Having all this data logic in the reducer simplifies the rest of the application and could make testing much easier as well: if reducers are functions, the test should assert that a certain input delivers the expected output. Reducers being functions means that the same rules apply to them regarding making their code clean, maintainable, and self-explanatory.

This app was improved for this thesis by migrating the code to TypeScript. This was my first attempt at such a task, so it involved lots of learning about the language concepts, technologies involved and how it applies to the stack used, but it was a great experience. I did not run into major issues, which is the ideal for a first try. Since the application is not very large and most of the complexity is confined to the data management and reducer implementation, typing did not present a big challenge. This, again, made for an engaging task. Some of the code was cleaned and made clearer, such as the back-end controllers which had name changes to better describe their purpose and the function in the fetch module bring broken into smaller ones. Some changes were also done in the front end, for example the utility module for calculating statistics which was much improved. Aliases were added to certain imports and unnecessary comments were removed to further improve readability.

3 Mobile Applications

On the second section of this portfolio, I will present two applications built for mobile devices. This is an area which we have covered in our studies as well, although not as extensively. The products on this section of the portfolio also highlight more extensively my own studies of the topics at hand.

Each day, more and more people access the web and digital services through their mobile devices instead of traditional desktop and laptop computers. According to StatCounter, as can be seen on Figure 13, market share based on web page views has been hovering slightly over 50% during recent years, with a noticeable increase to over 55% seen since October 2020.

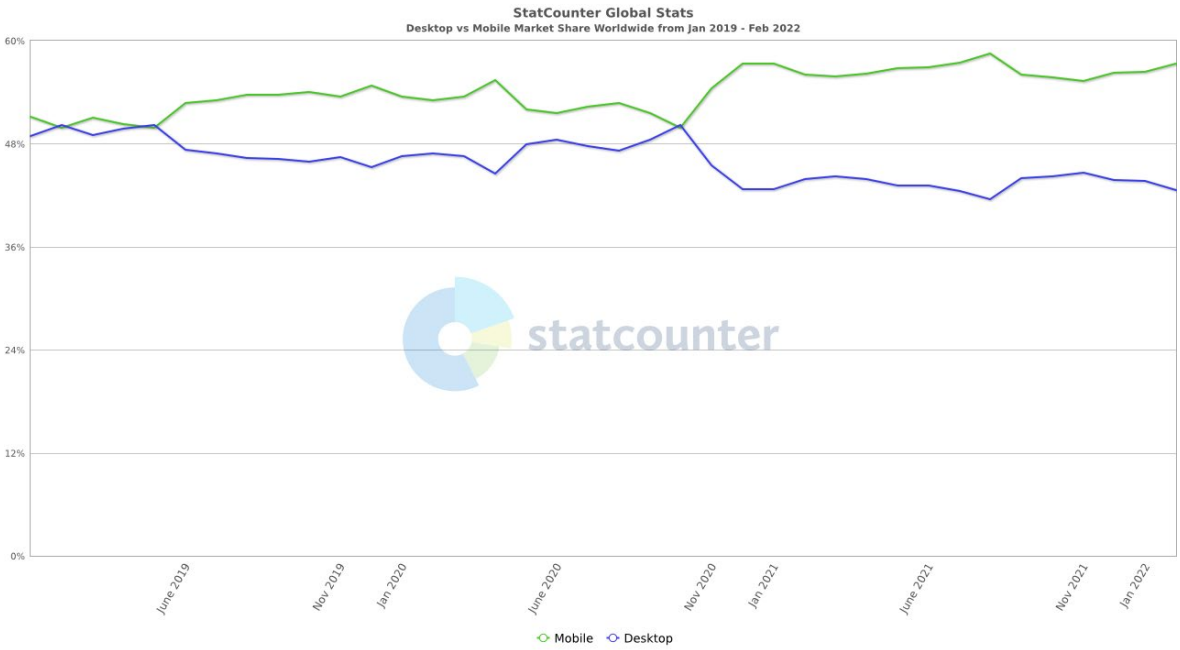

Figure 13. Desktop vs Mobile Market Share Worldwide (StatCounter 2022)

With this rise in usage from mobile devices, a new design paradigm has emerged during the last decade in web development, called responsive web design. A responsive website is one which painlessly adapts to different screen sizes; in other words, those who look correct regardless of whether they are being accessed from a desktop computer with a large screen, a tablet, or a smartphone. In order to do this efficiently, content needs to be reorganized as to provide a good experience, not simply shrinking or expanding the view, and some might be hidden altogether in smaller screens (Shade 2014). While responsive web applications can offer a good user experience to access content and features, traditional apps still have some advantages such as working better without an internet connection, the ability to send notifications even when the app itself is not in focus, support for many hardware features which might not be available in the browser and better overall

performance. The trade-off is that building an application is much more complex and demanding that merely ensuring an existing website is responsive. Web apps can have a more unified look, regardless of platform, while mobile apps should -ideally- feel more at home, respecting the design guidelines of the platform. Which is best regarding this visual aspect is more a matter of personal preference.

3.1 Theoretical Frameworks

The first aspect which can be discussed regarding mobile applications is how they run in the system. In both Android and iOS, applications run in a sandboxed environment. This means they cannot interact with each other and have limited access to the OS (Android Open Source Project 2022). This is done for security and privacy purposes, for example to avoid applications from seeing other apps' stored data, and to safeguard system stability. This applies to all applications, including those part of the core OS.

Another thing to keep in mind is that, while performance is also important as in web applications, in this case, the apps are intended to be run solely on mobile devices which are battery powered, meaning that execution efficiency and power usage are very important metrics to always pay attention to. An application which consumes too much battery power does not give a good user experience and it is very likely to be removed/uninstalled from the system if this is not fixed. Components which tend to use a lot of power are the CPU and GPU, the cellular and Wi-Fi antennae, and the GPS receiver. Some ways to address these is to minimize the amount of processing so that the cores can stay longer in idle, for example not performing unnecessary refreshes or updates, and only use radios when absolutely needed.

When developing a mobile app, more features are available than what is common in traditional computers, for example location services, access to both front and back facing cameras, biometric authentication (with fingerprint or face scanners), etc. User input is also different, not depending on movements with a mouse or trackpad and button clicks but on gestures, such as swiping, and screen presses. Mobile applications can, if required, use the same back-end structure as web applications. The back-end server can query the database, process data according to business needs and serve it through endpoints with an API. All this will be invisible for the mobile apps, which need only to bother with sending requests to the API and then displaying the data to the users. Back-end services can also be used for authentication/authorization, although this can also be done with cloud services.

Just like it was discussed before, there are both pros and cons to offering a responsive web application or a mobile app. There is, however, one third option which can be considered a middle ground between those, offering a compromise of sorts. These are cross-platform frameworks.

Cross-platform frameworks allow developers to build mobile applications which use the platforms' native APIs, but the application themselves are written with a platform-agnostic framework. The most well-known example of these is React Native which, just like regular React, uses JavaScript as programming language. Another popular option is Flutter, created by Google, which uses the Dart programming language. The biggest advantage of using a cross-platform framework is the number of resources required to build applications. If one app can run on multiple platforms, one team can work solely on that; in contrast, even considering only iOS and Android, that already doubles the number of applications needed to be developed when going the native route. It will also, most likely, require two teams, since development for both platforms is different enough, even down to the language used -JVM based on Android, such as Java or Kotlin and Objective C based on iOS, nowadays Swift- (Ilves 2022). Finding experienced developers with native apps can also be more difficult (and expensive), especially compared to JavaScript, which React Native uses and, as mentioned in the previous chapter, is very popular. Using a cross-platform framework can introduce a performance hit compared to a native solution from the added overhead. However, one of the biggest downsides from a user's standpoint is similar to what happens with web applications, only in this case is even more noticeable: the design of the user interface can feel out-of-place on the platform. Depending on the actual design, this can range from being mildly inconsistent with the native UI elements to completely out of place (common, for example, with apps using Android/Material Design-inspired UIs in iOS). Another is that even if the framework supports most features of the device, developers might not use some of them if they are exclusive to it (or to the OS, etc.) in order to keep feature-parity on all platforms, affecting negatively the user experience.

Once released, mobile applications are distributed through digital distribution channels such as the App Store on iOS and Google Play Store on Android. In some cases, this can be circumvented, but those are rare and can introduce security risks, which means the great majority of the users go through the official stores. The stores have requirements which application must meet in order to be allowed in, and submissions go through a review process before listing. The store operators handle the hosting of the applications. For paid applications, a percentage of the revenue is also taken from the sale. Also, although it can be avoided during development, developers must also have an account enabled to submit apps for publishing which has a fee for both platforms; that said, the one-time fee

of 25$ for a Google Developer account is much lower than the yearly 99$ fee for an Apple Developer one.

3.1.1 Multiplatform framework: React Native

I will begin tackling an example of a multiplatform framework, which I have chosen to do with React Native as it's the technology powering the first mobile product presented in this portfolio. That project was done as the final task for the Mobile Development course in Haaga-Helia where we used React Native as our main development framework.

With React Native we can build mobile apps using JavaScript (or TypeScript), which is the main programming language used during our studies (and this is probably the reason why we use it also for mobile). React Native uses React which makes it familiar to anyone who has already been working with the web library. They both feature -reusable- components, which in the case of React Native, invoke the corresponding native Android or iOS views, acting under the hood as native apps (React Native Contributors 2022a) although with some overhead. Data can be managed and passed with state and props, again familiar concepts to React developers. React Native's renderer, Fabric, bridges React logic and the host platform, by *creating* the tree, performing a *commit* -promoting the React Element Tree as the next to be mounted and scheduling a calculation of its layout information- and *mounting* into a Host View Tree (Kaszubowski & Lorber 2022).

React Native has other advantages. It is one of the most popular frameworks for cross-platform mobile app development, it is derived from React and it's also by Facebook. The React Native community is large, and it is easy to get help or find tutorials in order to further improve one's skills. Having a big, active community also translates to plenty of libraries being available to easily add certain functionality to our apps. These can be browsed in React Native Directory (at https://reactnative.directory/); the npm registry can also be used for finding libraries, but we must make sure they are compatible (React Native Contributors 2022b).

As part of our programming course, we have also used Expo. Expo is another framework, used for developing, building, and testing our apps, and can also be used for deployment. We can run apps in development mode to test features and debug, and in production mode to test their performance with a minified build. All of expo's features are present with their managed workflow, but developers are also free to generate native projects from their JavaScript code.

3.1.2 Native mobile development: iOS application in Swift

The second application presented on this section of the portfolio will be a native one, and since I use an iOS device myself, I decided to develop it using Swift.

Swift is a programming language from Apple, which came to light somewhat recently in 2014. Before it, the language for native application development on iOS (and MacOS) was objective-C. Swift is open source and contains plenty of documentation and guides on its official website. As a language, it puts emphasis on safety, with strict checks, and rules such as not allowing “nil” objects; performance, where it should be on par with C-based languages; clear syntax, intended to be a modern language which is constantly evolving (Swift.org 2022). The language is statically typed but types can be inferred, just as with TypeScript. Swift is not just intended for building mobile apps, it can be used to develop desktop applications as well. The IDE most often used for development in Swift is XCode, which is only available on MacOS. The software includes a simulator for mobile devices where apps can be tested without deploying to a physical one. It also features a graphical design module for building UIs.

Swift works with an application development environment for building iOS apps, as well as iPad, Watch and tvOS apps, called Cocoa Touch (Cocoa being the equivalent for MacOS). It allows developer to access OS features. Cocoa Touch is included in the iOS Software Development Kit -SDK- and contains frameworks, libraries, and APIs necessary to build our apps. One example of this is the Foundation framework, which is also included in regular Cocoa, providing “an object-oriented abstraction to the core elements of the operating system (Boudreaux 2009, Ch. 1). This includes, for example, file access, localization event dispatching and more. For developing the user interface, Cocoa Touch has included UIKit. However, Apple has introduced in recent years a framework called SwiftUI, which uses a declarative syntax somewhat reminiscing of (or perhaps inspired by) React. This makes building native UIs relatively simple, with clear code which is also in turn easier to maintain. Native elements and even animations can be added by simply describing what should be shown. UIKit and SwiftUI can be used together, if needed, which has been common while SwiftUI is still new and lacks all the features present in UIKit.

3.1.3 Project Management

While this application was developed on my own, it is based on and eventually became part of the project I took part in during the course “Multidisciplinary Software Project”. Being part of a team undertaking this project meant we also used non-development technologies, such as those related to project management. I will describe them briefly in this sub-section.

This project was done with the Agile methodology, using the Scrum framework during our development process, which focused on 2–3-week Sprints (depending on the semester's schedule).

At the beginning of each we would have a Sprint Planning meeting, where we discussed the customer's needs related to our tasks to-be-done in the backlog. Tasks were selected then to be worked on during that sprint based on their priority. At the end of each iteration, we would have two meetings: the Sprint Review, where our work was shown to the client in order for them to see the project's progress and give us feedback, and the Sprint Retrospective, where the team would look back on the work done during those past few weeks and try to learn and improve for the next one. Besides those, Scrum has also short daily meetings, of about 15 minutes each, called the Daily Scrum. Despite its name, we had those only during our lessons, so about 3 times per week, and working remotely because of the public health circumstances at the time, we had them over Microsoft Teams. At these meetings, sub-teams and individuals would shortly discuss what they had been working on and what they will continue working with. These meetings should not include longer discussions about the project or any parts of it.

Regarding the tasks and the backlog, we added there all the tasks which we planned to work on while building the project. There ranged from critical to completely optional ones and were always open to change or be even discarded altogether. At the same time, new features can be created as well, but they should only be selected to be worked on during the planning meetings for that sprint. The team is self-organizing which means its members create and choose which tasks to work on and how to solve the problems, including which technologies and tools they choose to work with. We regarded the tasks as completed when they met what is called the "definition of done", which is the criteria that the team determined necessary for the tasks to be deemed ready/complete. To manage our backlog, we have used the tool Trello by Atlassian. Trello features a board where the team can visualize the workflow with columns for categories, which can for example be "To-Do" or "In Backlog", "Selected for the current Spring", "In Progress", "Ready for review" and "Done". Each task can have an estimated allotted time, team members to whom it is assigned to, tags and more. Trello is based on Kanban-boards, which are a popular visual tool in project management.

3.2 Empirical part

This sub-section is the second empirical part of this portfolio and focuses on Mobile applications. As has been discussed, mobile apps and services are commonplace nowadays and keep expanding to both more and new aspects in our lives. The ability for a developer

to build one of them is very valuable and there are many differences from the process with web applications which warrant their own focus; or, in the case of this thesis, their own section.

I will describe what the applications are about, the development process -including learning involved- and the results as the application themselves. Part of the tasks include further refining of existing projects. I will aim to show the products at the ready state of a “minimum viable product”, that is one meeting the basic requirements or needs it is designed to fulfil.

The hardware used to develop these applications was the same as was used for the Full-Stack web apps, that is a MacBook Pro, 13-inch Mid-2014 model. The software used will be discussed under each project, but there were -thankfully- no major software incompatibilities found. In the case of the second product, being a native iOS app, working on a Mac was an advantage, as the tools for working with Swift receive more care and support for MacOS and building and testing is not supported on other Operating Systems.

The first application was started originally two years ago. Because of this, the refining which took place was a lengthier process for this application, considering both available technologies and the fact that my skills have developed significantly since then. This application was built as part of the Mobile Programming course of my studies in Haaga-Helia, as the final project and has been done entirely on my own.

The second application is more recent, done during my previous semester. Although not officially part of it, it was done to work with the system developed for our “Multidisciplinary Software Project” course. The project was done in collaboration with a mid-size Finnish IT company specializing in DevOps and in the end, we presented all the finished products, including my native iOS application and an Android app made with Flutter by another student in my team. This application was also well-received, and I have gotten good praise for my initiative by wanting to learn Swift on my own.

The source code for these applications is available online and links are provided in this thesis. As they are not deployed to any stores at the moment, it is not possible to test them except for building them from source, meaning I cannot include any links to production versions (since it was part of a school project, the backend is also not available anymore for the second app, although it can also be run locally). I will add screen captures from the working products to show how the applications look in action while describing them.

3.2.1 Password Generator App

At the end of the Mobile Programming course, which I took two years before the writing of this thesis -in April 2020-, was a final project which encompassed all we had hitherto learnt. As part of this project, we had to develop our own full-fledged mobile application. This application should use the technologies covered during the course and include additional capabilities and features which we researched on our own. Based on these premises, my idea was to develop an application to generate random, secure passwords for the user with the option to save them for reminding. This idea felt to me could help me meet all the requirements for the project, as well as being an interesting (and realistically useful) topic. The source code for the application can be found here:

https://github.com/ernven/Mobile-Password-Generator

Based on the tech stack used through the course, the application was built using React Native. The project was built using Expo, the framework for building these types of apps which we used during the course. Development of the application took about 10 days, not including testing and documenting. The initial version was well received by the teacher, especially about the extra features implemented.

Moving onto the implementation, figures 14 and 15 show the main screens for the app.

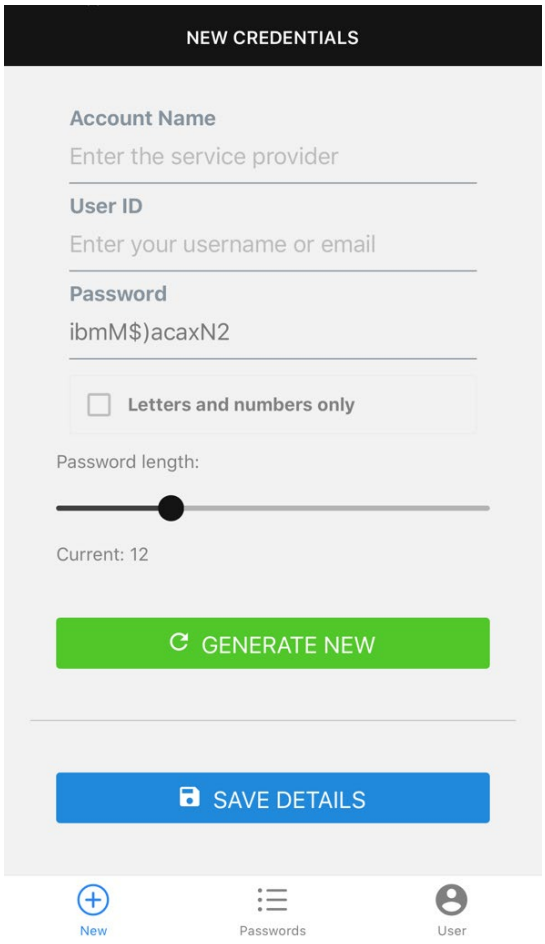

Figure 14. New account screen

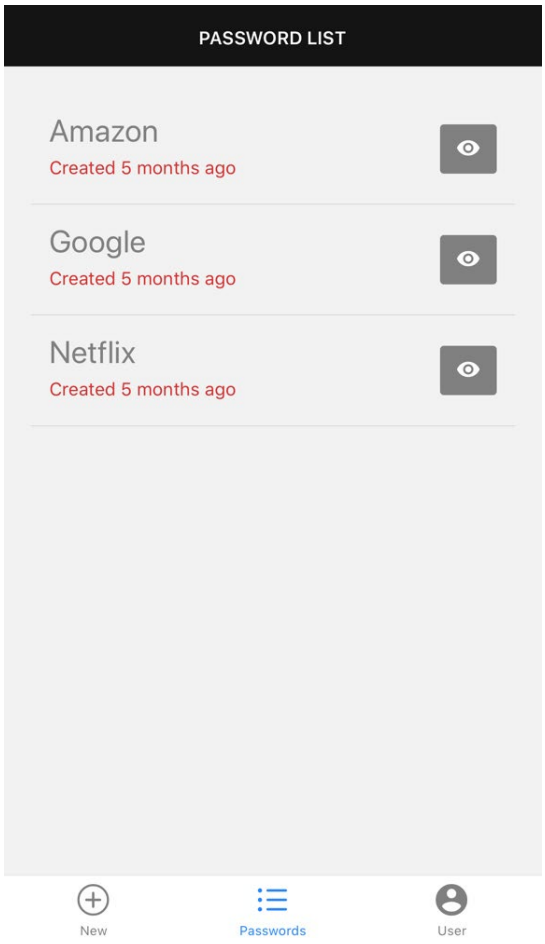

Figure 15. Account List screen

First, in Figure 14, we see the main screen where we can generate a new random password, according to the settings which we can configure to our liking. We can also add an account name and username to be saved along with the password. To show how an application works with data from a remote API, instead of generating the password on-device, they are fetched from online sources. I had found two open APIs which can be used for this, and they are named in the documentation. The one I used for testing and demoing the product requires a sign up to get a personal token in order to be able to make requests to it, but it is a simple process and the limits for requests are enough for personal uses such as this one. Once a password has been saved, it is added to the list shown on the second screen (Figure 15). The account names are displayed, along with a helpful text showing how long it has been since their creation for users to keep track of. By tapping the buttons on the right, the whole account info is displayed, and entries can also be deleted. These are not stored locally, but on a remote database server so they will persist even if the user deletes the application. For this I used Google´s Firebase, which we had used during this course, so I could get more experienced with it, especially with handling different sets of data. Firebase has decent documentation which can be followed on their website for reference.

Next are some of the new features which I learnt specifically for this project.

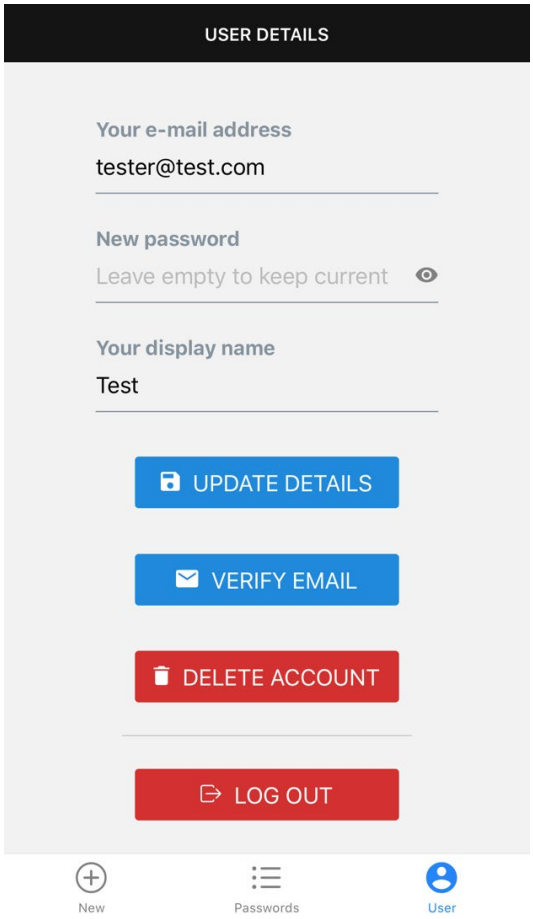

Figure 16. User Details screen

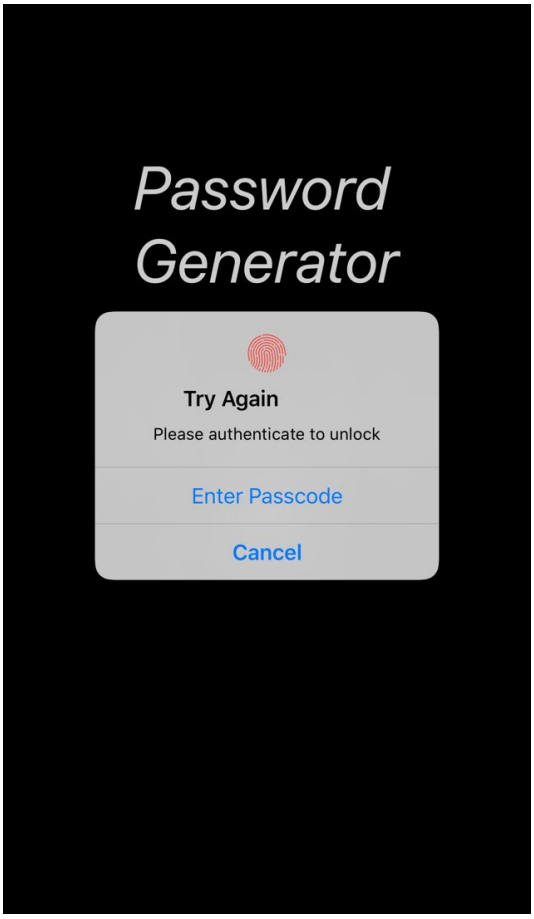

Figure 17. Local authentication screen

User data such as account details could be safely saved in the application's documents, since these are kept out of reach from other apps as applications work in their own sandboxed environment. However, based on the original idea of this project, it made sense for users to have different accounts so that the information could be persisted somewhere in case the user would remove the application or get a new phone, to name a few examples. Since I had already been using Firebase as a cloud database solution and they also offer an authentication service, it seemed natural to implement that. It is included in the Firebase package, so all that is required is to import the necessary modules. In Figure 16, the User Details screen can be seen and from there can be described the functionality provided by Firebase. Users can be created with an email address and password -there is more options, for example social logins, email links, etc. but those are not implemented-; the email address can be verified, which could be required for certain capabilities, if desired; the user can log out or delete their account altogether. For sensitive changes, such as deleting an account or changing the password, we can require the user to login again, if they haven't done so recently. All these features can be implemented thanks to Firebase with relative ease. The authorization module contains functions and properties which can be used to implement these (in v9 available with different imports).

The second feature which I thought would make sense, given the nature of the app, is to have local authentication as well, which would "lock" the application if it went into the background, for example if the user switched to another app or locked their phone. This way, if a user were to pass their phone to someone, they wouldn't be able to switch back to the password app and see sensitive information. The best aspect of this feature is, in my opinion, that thanks to biometric authentication, the whole process can be made -almost- seamlessly; if one had to enter a pin code every time, it would be more prone to causing an annoyance to the user, but if it is simply scanning your fingerprint -which is already stored by the system- or your face, it's both quick and effortless. This is basically the main selling point about adding biometric authentication in general. I researched this and found that expo already includes a module to enable such capability, aptly named expo-local-authentication. This module, like many in expo and React Native, features an API to use the native capabilities of the system, which means the application does not see, store or in any way process the fingerprint, it is done by the hardware on the device and the platform software, ensuring that the user is provided with the best security and reliability with their biometric details and the safeguarding of the application data. The result can be seen in the screenshot shown in Figure 17, looking like any other prompt for TouchID on iOS (the app was tested on iOS).

One more thing to note is that validation has been added to the input fields, so the app will check, for example, if a required field is kept empty. Firebase also returns error in cases such as an email address already in use, wrong password and more. All these are considered, and feedback is passed to the user through helper text along the text fields so that they can be aware when something goes wrong, why it happened and what should they do to address the problem. I believe this is also a very helpful feature which should always be present, albeit not always easy to implement, especially when there are many possibilities for things going wrong.

As part of this thesis project, I have also tried to improve and refine these existing products in order to show the best possible version of them. For this particular case, the improvement phase took the longest time and most intense work. The reason for this seems very clear to me; this application is the oldest in the portfolio and was done in what could be considered the “middle” of my learning journey at Haaga-Helia. I had learnt a lot by then but looking back, I can see now how much more there was to improve (at the same time, I can foresee how much I will have to keep learning as I go on, even in working life). I still think this product is a valuable addition to my portfolio, not just because of the application itself, but also because it showcases this journey, which I believe is important considering the work it is now part of. Also, because it allows me to look back onto my own work with a critical eye, point out specific issues with it and show how I am now able to improve them.

This refining phase also made something much clearer for me. As I have mentioned in the theoretical portion, writing clean and maintainable code is of the upmost importance, in order to be able to fix issues with it and to enable others to work with it without much trouble and frustration. Another important aspect for it, however, is regarding upgradability and potential migrations. While improving this application and updating its components, it has become evident to me that one of React’s -and JavaScript in general- advantages, that of its massive library of components available to easily add features and capabilities can also be a double-edged sword. As time passes, these components get updated and some versions break compatibility with previous ones, which requires bigger reworks of the implementation. If the base code is messy or poorly written, what could be an easy fix taking from a few minutes to a couple of hours can easily turn into a nightmare which takes from days to weeks. In a commercial project, this increases the costs by a large margin. The same can happen not just with component updates but if, inside a project, new features would have to be added. Or something as simple as fixing errors or security flaws. The worst aspect of this is not when the work becomes much more complicated or expensive, but when because of that it gets ignored. When thinking about software, users expect updates and while their willingness to pay for it can be argued, in case of bugs, errors or -

even more pressing- security flaws, digital products and services are expected, as they should be, to be free of them.

Some of the changes done as part of the improvement work were:

- States which were related were changed to one State object encompassing all. For example: all user details can be grouped into one state.
- Long functions have been made shorter by extracting parts of them. The names were generally good enough and self-explanatory but, for example, “save user” can simply handle the saving -as the name says- and “handle save success” can deal with a successful operation instead of being all grouped into one.
- React Native tends to have long JSX returns with many components and nesting. To relieve this somehow, new custom, reusable components were made. The best example of this is the form components used for login/sign-up screens.
- Related to the previous, styles have been moved into their own StyleSheet reference using the provided API. This clears most styles from the JSX, taking them out of the main component altogether. This also works well when extracting components, as the styling repetition can also be avoided. Some inline styling has been left in cases where they are unobtrusive, to keep the StyleSheets neat as well.

Upgrading to the latest Expo SDK and React Native, as well as all the other components required quite a bit of work. For the most part, after reimplementing all the features, the changes should be invisible to the user, meaning the app should work in the same way it did before. However, because of changes to UI components and to elements regarding the native interface (for example, changes to dark mode, screens including camera “notches” now, etc.) some visual differences were introduced in order to keep the work and deadlines under control and, at the same time, improving the overall look of the app.

I will now give some more concrete examples about these improvements, using as well visual aids. These were all taken from the component UserNew.js. The next couple of figures show how can states be improved by grouping them when they are related. Figure 18 represents the code for the first version of the app and Figure 19 is the improved one.

```
export default function UserNew() {
    const [email, setEmail] = useState('');
    const [password, setPassword] = useState('');
    const [emailError, setEmailError] = useState('');
    const [passwordError, setPasswordError] = useState('');
    const [visible, setVisible] = useState(false);
```

Figure 18. State properties for UserNew in the first version of the app

```
export default function UserNew() {
  const [visible, setVisible] = useState(false);
  const [userDetails, setUserDetails] = useState({email: '', password: ''});
  const [errors, setErrors] = useState({emailError: '', passwordError: ''});
```

Figure 19. State properties in the new version of the app

As we can see, the email and password are part of user details, or could also be called credentials. The same can be said about having individual states for each error. Having these separated clutters up the code and brings no advantage, except maybe making them slightly easier to update, but at the same time making keeping up with state values and updates more complicated.

Next is shown how a function can be made cleaner and easier to work with.

```
const signUp = () => {
    setEmailError('');
    setPasswordError('');
    firebaseAuth.createUserWithEmailAndPassword(email, password)
    .then(() => {
        firebaseAuth.currentUser.sendEmailVerification()
        .catch((error) => {
            Alert.alert("An error occurred: " + error);
        });
    })
    .catch((error) => {
        if (error.message.includes("formatted")) {
            setEmailError("The email address is not valid");
        } else if (error.message.includes("6 characters")) {
            setPasswordError("Password must be at least 6 characters long");
        } else if (error.message.includes("already")) {
            setEmailError("Email address already in use");
        } else {
            Alert.alert(error.code + ": " + error.message);
        }
    });
};
```

Figure 20. Sign Up handler function v1

```
const handleSignUp = () => {
  setErrors({emailError: '', passwordError: ''});
  createUserWithEmailAndPassword(auth, userDetails.email, userDetails.password)
    .then(() => sendEmailVerification(auth.currentUser))
    .catch(error => setErrors(handleError(error.message)));
};
```

Figure 21. Sign Up handler function v2

As can be seen, even disregarding the changes to the Firebase syntax due to upgrading to the latest version, the differences are quite drastic. The function went from being very

long to just a couple of lines. Again, from improving our state management, it is now easier to reset the values for errors together, in one line. In fairness, the error handling part did not simply vanish into thin air, it was moved into its own function. This makes sense, however, since the function is supposed to handle the sign-up process, not error messages. It would not be terrible if we could catch a simple error and show a generic message, but here we had and if-else with four different outcomes. Moving into its own function makes more sense, makes this function easier to read and, best of all, since it was actually moved into its own module inside the “utils” folder, not only we can import and call the function in other components of the app -which has been done- but it could be ported as well to any other app which uses authorization with Firebase! This is a major advantage when thinking about simplifying tasks, improving maintainability, and following the “don’t repeat yourself” principle.

And finally, here is the look of the new version of the app. Functionality-wise remains the same as the previous, but it has updated libraries, improved reliability, and the look has been tweaked to have a more native feel. In these screenshots we can also see some of the feedback presented to the user.

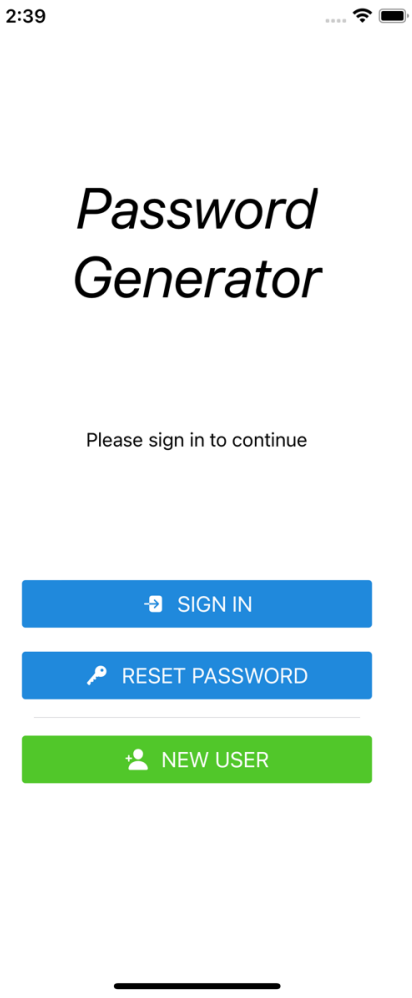

Figure 22. Main screen of the app

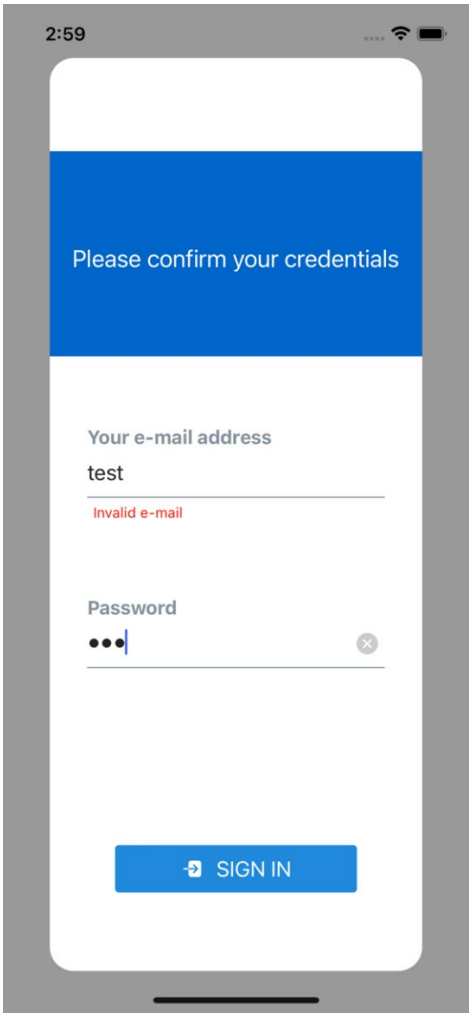

Figure 23. Login screen with helper text

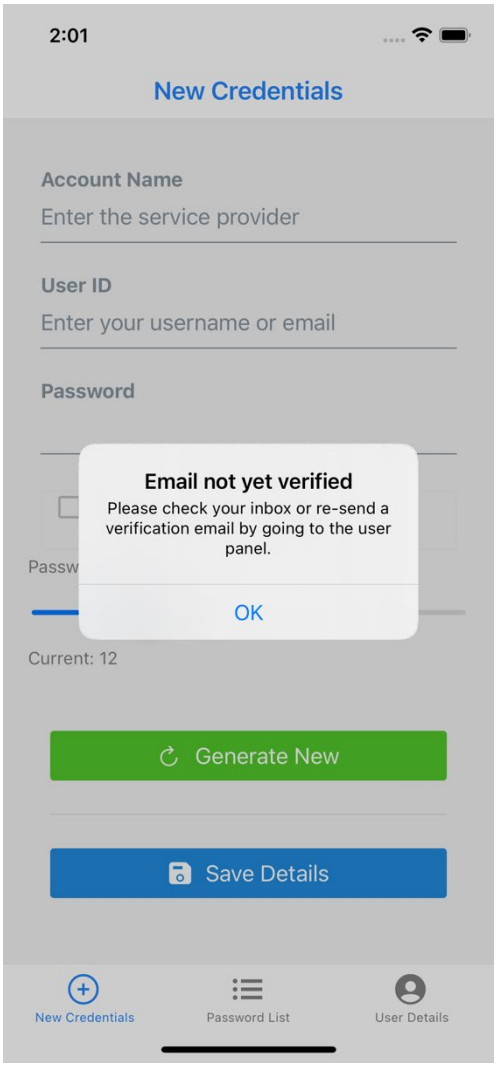

Figure 24. Main screen with alert

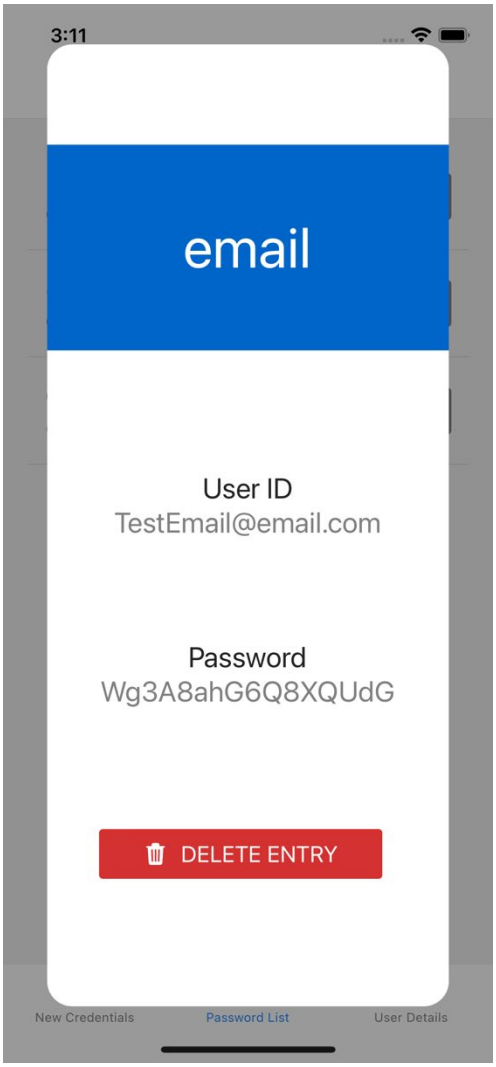

Figure 25. Account details overlay card

3.2.2 Pop-Up Rooms Project iOS App

The background for this app is that I had been for a long time interested in learning native iOS development and Swift programming. As I mentioned, people are using more and more their mobile devices, me included, and having been an iPhone user since Microsoft gave up on Windows Phone, I feel right at home in the platform. By using more and more my phone and other portable devices, and being enthusiastic about technology, I love the things which can be made true with the right software to go along, and I feel that nothing can beat a native experience for performance, feel and integration. I haven't had time to actually learn it while focusing on my courses until last semester. I had only one programming course left, Multidisciplinary Software Project, which meant that I had some extra time to consider new technologies. When the intensive week came, I decided not to take any extra courses and instead focus on finally learning iOS development on my own. I found great tutorials and documentation on Apple's developer website (a bit to my sur-

prise!) and embarked in my journey. Only that, instead of following some random examples, I decided to base my learning work on the project which we were doing as part of the course. This way I would feel to be building something useful, and potentially -if everything worked out- might even be able to add it to the project. The app turned out quite nice, even if it was not extremely complex; it met the requirements stipulated in the project and had a good UI and UX to go with it, and I also received good feedback from my teammates and from the person representing the client company. As with the other products, the source code is available on GitHub. In this case, at https://github.com/ernven/popup-rooms-ios

The project itself was based on a set of “pop up” meeting rooms which the client had at their office. These rooms work like regular meeting rooms, for in-person or virtual meetings, but they require no booking or reservation, and can be occupied whenever found available. While this is good for simplifying the process of getting a room, it also adds the problem of actually finding an empty room when you don’t have any fixed schedule. This is where we came in. The task was to build a system based on IoT devices to detect whether a room was occupied and if so, communicate that somehow to a web client which would show a list of rooms and their status. I was part of the frontend team which worked on the web app written in React, and since I worked on the web client for the users, I figured a mobile app would not be completely alien to me. I asked my colleagues if anyone was interested in building the app with me, but they did not seem very eager to. One of them did build an app with Flutter and Dart, which became our team’s Android app (to complement my iOS app). Since I have built this app on my own, I feel it makes a good addition to this portfolio.

The application is simple from the users’ perspective: when opened, the main screen shows the list of rooms and their status, as can be seen in Figure 26. The list can be refreshed on appear and manually by swiping down (as it’s common behaviour). Both room data and status are fetched from our backend, so without it there will be no information displayed. Rooms can be filtered by available only, just as in the web client. The mobile app adds one new feature which is not present in the web version; the ability to “star” or favourite certain rooms, and to see only those (or, by combining both filters, only starred available rooms). This feature made more sense to be implemented on mobile because we did not have user accounts to persist data, and on mobile we could at least store user data into the application document directory, where it would reside until the application was deleted (potentially, it could even be synched to iCloud). On the web version, even though we stored some data in Local Storage, such as the filters, and those remain even after closing the browser, they are lost if the local data is deleted, for example when clearing the browser’s cache files.

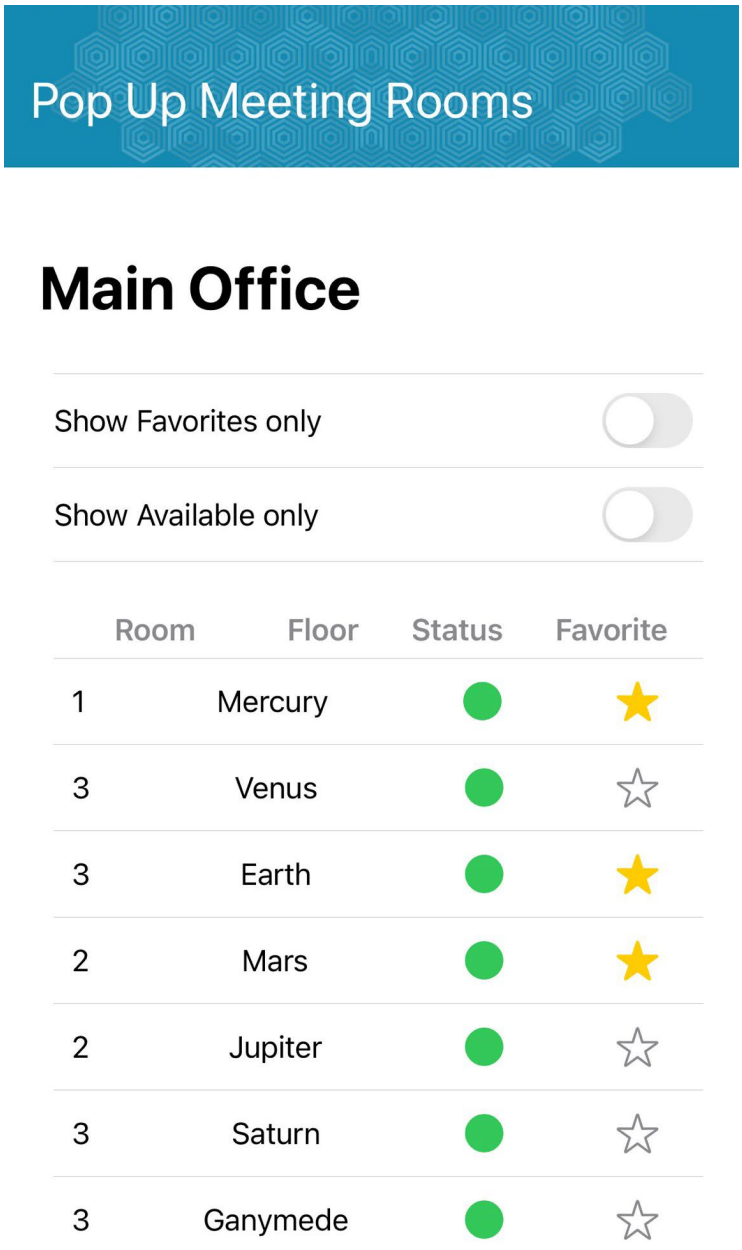

Figure 26. Main screen of the app

Thanks to SwiftUI, building the user interface does not take a lot of work nor time, and native-looking and aesthetically pleasing interfaces can be built without much trouble, as the example shows. The visual design is based on the one chosen for the web application but applied to native iOS elements and the platform guidelines.

During the second to last week, I also implemented a widget for quick checking the rooms' status. Widgets can be added to the home screen, and they come in three sizes. Only favourited rooms will show on the widgets. The size also impacts the amount of information which is shown. Two widget sizes, Large and Medium are shown on Figures 27 and 28, respectively. When adding widgets, they will show using some predefined sample data, for quicker loading, but widgets can also use dynamic data which the app provides.

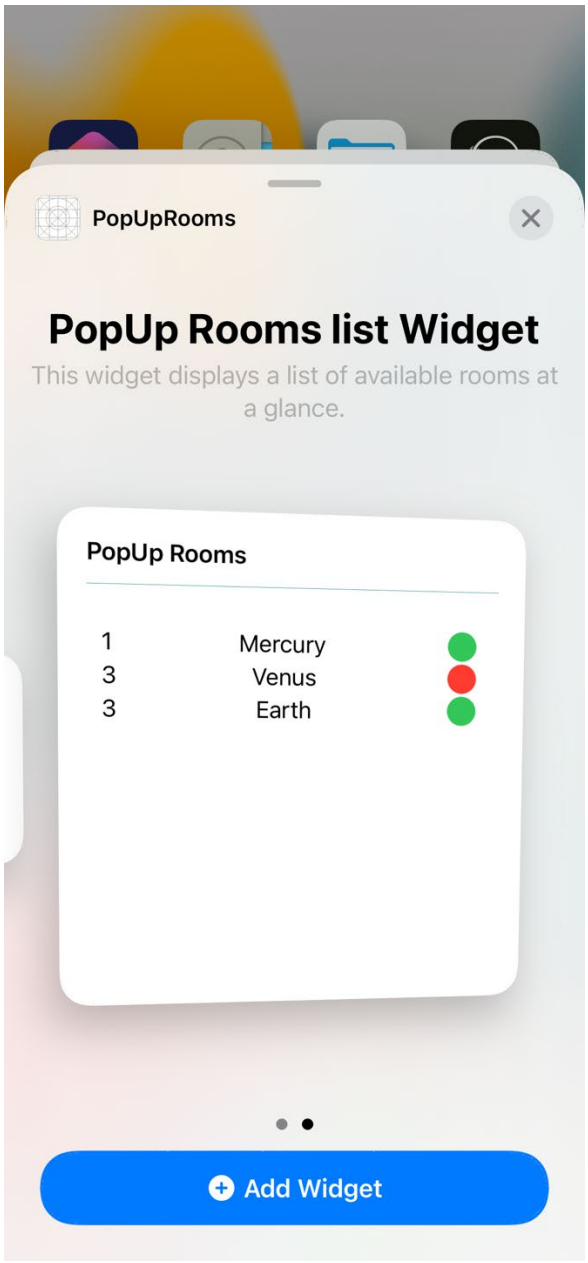

Figure 27. Widget size L

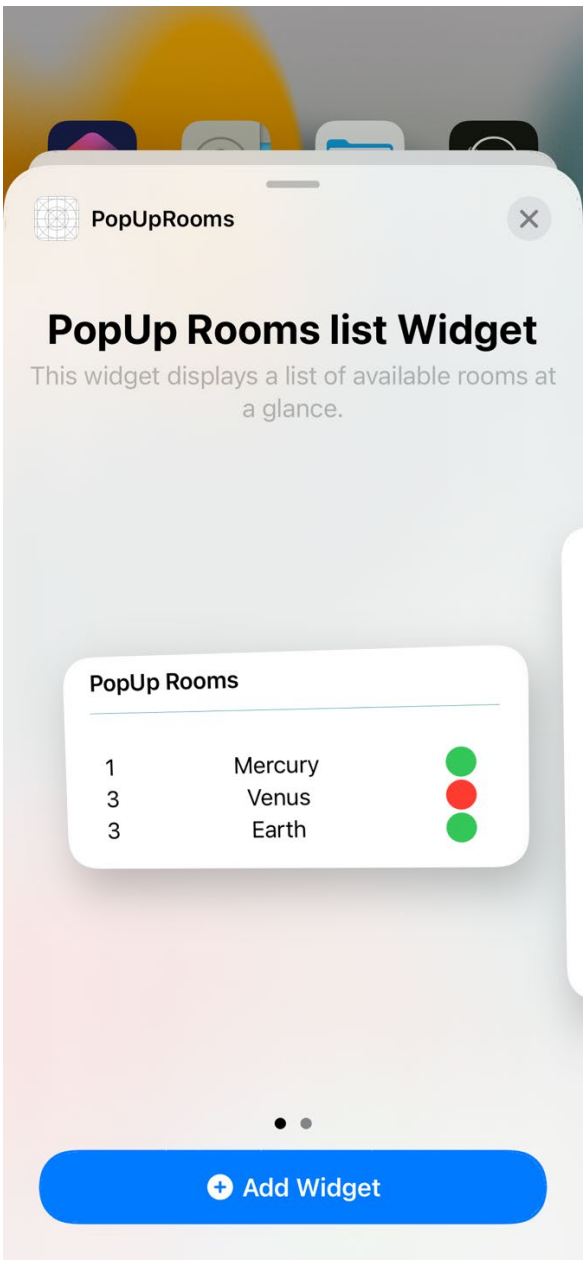

Figure 28. Widget size M

They are a useful feature and ended up working well, but they took a lot of work and were more complex to set up than I initially thought, to the extent that I was not sure whether I would be able to finish on time. Widgets are an extension to the app. An app can have different ones even, displaying different sets of information, presenting different layouts, etc. They could be customizable or not. Their handling of the data is a bit complicated to start with. Because widgets are meant to be light on system resources, and particularly the battery, they do not update in the same way as the application. The content of a widget updates based on a “timeline”. A timeline entry has a date (with time) based on which it should update the content. We create widgets with sample data for previews for faster loading and placeholder widgets to show while loading, along our regular ones. A useful aspect of the timeline is, in the case of this app, we can refresh the widget content only during office hours, which is when users might need the pop-up rooms.

The approach to building applications in Swift is different from using JavaScript and frameworks like React (or Native, on mobile). The language and project structure are reminiscent of other object-oriented languages, such as Java or C#, perhaps more to the latter. In the application is defined a Model for the Room objects which includes all the property fields and a custom init, as some properties do not match the objects coming from the API, for example whether the room is favourited which is an iOS-only feature. In the Model Data file (here RoomData.swift) the data handling logic is located. The app worked initially with sample data from a file, but this was later replaced with fetching data from the project's backend. There are also methods for loading and saving favourites to a file for persistence. This is not a very complex process but fetching data from the API is much more convoluted than in JavaScript, where it only takes a few lines of code. The UI portion shows more similarities, especially with React, since the introduction of SwiftUI. Views are located under the folder of the same name. Custom View component may contain some application logic, which is located at the top, and the actual View elements under "body". Elements can also be grouped into "stacks", which can be Horizontal or Vertical. Some UI elements like the mentioned stacks, List or Navigation Views are provided by the library, and modifiers can be applied to elements to change their look, behaviour or add effects such as resizing or animations. SwiftUI also features the concept of state, for a single source of truth, which is managed by the system in order to keep track of changes and manage the updating of the views as necessary. To allow the Views to change data stored in the state property, binding is used, passing the variable name with the $ prefix operator. Similar rules from React regarding state also apply here, such as for it to be located in the last common ancestor, or highest component in the tree which uses the property.

4 Discussion

As I reach the last chapter of this thesis, I can look back on the work I have set to do. Considering the objectives which I had defined for it, I can claim that I have met them, by having conducted strong research of the theoretical background, presented the applications in working condition along with the improvements made upon them, and now reflecting on my learning journey.

The outcome of the research is a good, approachable look at the technologies used while taking into account their status in the software development world, their history and advantages that they offer in their particular fields of use. Conducting it has allowed me to continue learning and obtain a deeper knowledge of the subjects involved, which I am happy for, as it will be valuable while I continue my professional journey, helping me also in working life. The research has comprised the whole stack of a web application, from the database technologies, through the backend into the front-facing client which will be the point of interaction with the users. I started with a rather solid base of knowledge, built during my courses in Haaga-Helia, but I had now the opportunity to go one step further. One example of this is learning about TypeScript which, while new for me, stands on top of the knowledge of JavaScript. The research included also its own separate section for mobile technologies, where it has been discussed both paradigms of mobile development, multi-platform frameworks and native solutions. Regarding these, I can understand it might also be dependent on the scope of the project, as well as the means and resources of the company, but in the end, a native solution is always the superior choice, if it can be afforded. Special mention should go to the sub-chapter devoted to best practices of writing software; this is one area which I can say I have learnt extensively, if not the most, and I believe I must continue to develop in this subject in the future.

Regarding the portfolio itself, and the presentation of the selected products, I am also very satisfied with the results. The applications are not just working and include the desired functionality, meeting the requirements initially stipulated including those regarding the usability. They have also all been refined as part of this thesis work and the improvements have been successful. This task took some time longer than I initially expected and, at certain times, took a significant amount of work. This was the case especially for the oldest application in the portfolio, the Password Generator mobile app. This refining process, however, has allowed me to also learn about what goes on with project maintenance and what to expect when software gets outdated and needs to be brought up to speed. Some remarks were already written in the pertinent sub-chapter, but it is good to point out that maintainability (which goes hand-in-hand with reliability and security, when talking about keeping an app up to date) is also another oft-overlooked topic, which can be surprising

considering the widespread presence of agile practices focusing on constant improvement. One last thing to note is, that this thesis's work has allowed me to continue practicing the essential programming skill of problem solving, in this case with the refining of the apps, also by looking for new solutions to problems found. One example of these is adapting the Password Generator app so that UI components can be more independent. In order to do this, it was necessary to re-analyze the component structure and extract redundant elements into new, smaller ones, improving their reusability.

During this project, I have been able to keep my schedule in check and meet my deadlines, even if some were rather tight at times.

4.1 Final Remarks

One, if not the most, important aspect of a work of this kind is the learning process. While working developing the apps and researching the relevant topics, I have not only dug deeper in the knowledge of software development (in particular, web and mobile) but I have also learnt a lot about making better applications. By better, I mean taking particular care about the code being written; that it is clear, concise, functional, and easy to understand. It should not contain parts which do not belong there or are unnecessary. All this will make it easier for others to understand our programs, which is unavoidable in a professional setting. Since I am talking about doing software engineering in a professional manner, it is also important that the tools we use are the correct ones and relevant. The world of technology moves fast, and it takes extra effort to keep up, but that we must do if we don't want to be left behind. Speaking of changes, and in the ever-improving spirit of Agile, developers themselves should always strive to continue learning, growing as professionals, and constantly improving. This is not just about learning new frameworks, programming languages or tools, but also working on our problem-solving skills and applying our knowledge to the specific requirements of a project or a client. I have gotten feedback from colleagues regarding the applications, and it has helped me to keep constantly improving on this.

Looking back on the work done, I consider I have reached a good outcome with the research, but perhaps the strongest point is the portfolio itself, where my own skills are shown, and the result of the improvements on the applications. I am happy with each version presented here as all parts make up for a varied yet balanced software development portfolio.

Regarding further improvements, which sadly fell out of the scope of this thesis, the biggest improvement which could be implemented to these projects is automated testing.

First and foremost, because it is, from my own experience, an almost required skill in today's world of software engineering, and one which -sadly- we have not learnt much about during our studies. Automated testing can be an extremely helpful tool when continuously improving an application, as adding new features or reworking older ones can lead to breaking parts of the software, and manually testing for these issues can be time consuming and inefficient, as well as risky if something were to slip through the cracks. On this topic, there is also the concept of Test-Driven Development, which consists of writing the tests *first*, before writing any line of code for the application. I would have liked to have the chance to tackle this -or at least begin to- as part of this thesis work, but it remains among the top subjects I will strive to learn and practice in the near future.

I hope this thesis can be a valuable resource for fellow students who find themselves on a journey like mine, that it can help them complement the knowledge acquired during courses and continue learning new topics, as it has helped me while writing it.

References

Android Open Source Project 2022. Application Sandbox. URL: https://source.android.com/security/app-sandbox. Accessed: 27 March 2022.

Ashdown, L., Keesling, D. & Kyte, T. 2021. Oracle Database Database Concepts, 21c. Oracle. URL: https://docs.oracle.com/en/database/oracle/oracle-database/21/cncpt/database-concepts.pdf. Accessed: 10 March 2022.

Beal, V. 2021. Acronym Guide To Web Stacks. Webopedia. URL: https://www.webopedia.com/reference/webstack-acronyms/. Accessed: 11 March 2022.

Boudreaux 2009. Programming the iPhone User Experience. O'Reilly Media, Inc. Accessed: 28 March 2022.

Chacon, S & Straub, B 2014. Pro Git. 2nd Ed. Apress. E-book. URL: https://git-scm.com/book/en/v2. Accessed: 20 March 2022.

Chamberlin, D. D. & Boyce, R. F. 1974. SEQUEL: A structured English query language. SIGFIDET '74: Proceedings of the 1974 ACM SIGFIDET workshop on Data description, access and control, pp. 249-264. URL: https://dl.acm.org/doi/10.1145/800296.811515. Accessed: 28 April 2022.

Codd, E. F. 1970. A relational model of data for large shared data banks. Communications of the ACM, 13, 6, pp. 377-387. URL: https://dl.acm.org/doi/10.1145/362384.362685. Accessed: 27 April 2022.

Codeacademy 2022. Back-End Architecture. URL: https://www.codecademy.com/article/back-end-architecture. Accessed: 12 March 2022.

Fielding, R. T. 2000. Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation. University of California, Irvine. URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf. Accessed: 21 March 2022.

Git 2022. Reference: Guidelines. URL: https://git-scm.com/docs/SubmittingPatches. Accessed: 20 March 2022.

Heroku 2022. Security Assessments and Compliance. URL: https://www.heroku.com/policy/security. Accessed: 25 March 2022.

Heroku Dev Center 2022. Choosing the Right Heroku Postgres Plan. URL: https://devcenter.heroku.com/articles/heroku-postgres-plans. Accessed: 25 March 2022.

IBM 2019. Relational Databases Explained. IBM Cloud Learn Hub. URL: https://www.ibm.com/cloud/learn/relational-databases. Accessed: 10 March 2022.

Ilves, K. 2022. Introduction to React Native. University of Helsinki. URL: https://fullstackopen.com/en/part10/introduction_to_react_native. Accessed: 26 March 2022.

Kaszubowski, B. & Lorber, S. 2022. Render, Commit and Mount. URL: https://reactnative.dev/architecture/render-pipeline. Accessed: 27 March 2022.

Martin, R. C. 2009. Clean Code: A Handbook of Agile Software Craftmanship. Pearson Education. E-book. Accessed: 20 March 2022.

Microsoft 2022. App service overview. Azure Docs. URL: https://docs.microsoft.com/en-us/azure/app-service/overview. Accessed: 25 March 2022.

MongoDB 2021. Understanding the Different Types of NoSQL databases. URL: https://www.mongodb.com/scale/types-of-nosql-databases. Accessed: 10 March 2022.

Mozilla Contributors 2022. The Websocket API (Websockets). URL: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API. Accessed: 11 April 2022.

Nguyen, K. A. 2009. Database System Concepts. OpenStax CNX. URL: http://cnx.org/contents/b57b8760-6898-469d-a0f7-06e0537f6817@1. Accessed: 10 March 2022.

Node.js 2022. Introduction to Node.js. URL: https://nodejs.dev/learn/introduction-to-nodejs. Accessed: 22 March 2022.

React 2022. Reconciliation. URL: https://reactjs.org/docs/reconciliation.html. Accessed: 25 March 2022.

React Native Contributors 2022a. Core Components and Native Components. https://reactnative.dev/docs/intro-react-native-components. Accessed: 27 March 2022.

React Native Contributors 2022b. Using Libraries. URL: https://reactnative.dev/docs/libraries. Accessed: 27 March 2022.

Reaktor 2022. Assignment 2022: Developers. URL: https://web.archive.org/web/20220221122400/https://www.reaktor.com/assignment-2022-developers/. Accessed: 11 April 2022.

RedHat 2020. What is a REST API?. URL: https://www.redhat.com/en/topics/api/what-is-a-rest-api. Accessed: 21 March 2022.

Schade, A. 2014. Responsive Web Design (RWD) and User Experience. Nielsen Norman Group. URL: https://www.nngroup.com/articles/responsive-web-design-definition/. Accessed: 26 March 22.

Smith, S. 2022. Architecting Modern Web Applications with ASP.NET Core and Microsoft Azure. Microsoft. E-book. URL: https://aka.ms/webappebook. Accessed: 24 March 2022.

Stack Overflow 2021. Annual Developer Survey. URL: https://insights.stackoverflow.com/survey/2021. Accessed: 9 March 2022.

StatCounter 2022. Desktop vs Mobile vs Tablet Market Share Worldwide. StatCounter GlobalStats. URL: https://gs.statcounter.com/platform-market-share/desktop-mobile/worldwide/#monthly-201901-202202. Accessed: 26 March 2022.

Swift.org 2022. About Swift. URL: https://www.swift.org/about/. Accessed: 27 March 2022.

Torppa, T., Peuraniemi, T. & Rapo, J. 2022. TypeScript: Background and introduction. University of Helsinki. URL: https://fullstackopen.com/en/part9/background_and_introduction. Accessed: 9 March 2022.

TypeScript Docs 2022. TS for the New Programmer. URL: https://www.typescriptlang.org/docs/handbook/typescript-from-scratch.html. Accessed: 9 March 2022.

Vainio, N. & Valleala, A. 2021. Do’s and Dont’s of Dev Academy Pre-Assignments. Solita. URL: https://dev.solita.fi/2021/11/04/how-to-pre-assignments.html. Accessed: 26 March 2022.