Bachelor’s thesis

Information and Communications Technology

2022

Henri Kestiö

WEB APPLICATION DEVELOPMENT: WHAT DOES IT CONSIST OF?

TURKU AMK

TURKU UNIVERSITY OF APPLIED SCIENCES

BACHELOR'S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Information and Communications Technology

2022 | 34 pages, 26 pages in appendices

Henri Kestiö

WEB APPLICATION DEVELOPMENT: WHAT DOES IT CONSIST OF?

As technologies evolve and web application requirements grow more complex, it would be more apt to describe application development as something similar to building an apartment building where multiple domains of expertise need to be combined. This thesis explores the major domains of modern web application development and describes the process of building a full-stack application using current technologies to illustrate the different areas in a more practical way. Various internet resources about web application development were studied for this purpose. A full-stack web application was developed, setup online and the source code was stored to a public GitHub repository.

The main results of this thesis were a summarized overview of each of the major areas of web application development and a process description of the development of a full-stack web application. The results show that while web application development can be broken down into a few major areas, such as frontend and backend development, it is clear that each area requires a depth of knowledge that also needs to be constantly updated and expanded on as technologies evolve. This reveals the need for learning resources that are also constantly evolving.

KEYWORDS:

Web development, databases, application architecture, DevOps, full-stack

OPINNÄYTETYÖ (AMK) | TIIVISTELMÄ

TURUN AMMATTIKORKEAKOULU

Tieto- ja viestintätekniikka

2022 | 34 sivua, 26 liitesivua

Henri Kestiö

WEB-SOVELLUSTEN KEHITTÄMINEN: MISTÄ SE KOOSTUU?

Teknologioiden kehittyessä ja sovellusvaatimusten kasvaessa monimutkaisemmiksi, olisi sopivampaa kuvata sovelluskehitystä kerrostalon rakentamisen kaltaiseksi projektiksi, jossa on yhdistettävä useita osaamisalueita. Tämä opinnäytetyö tarkastelee nykyaikaisen web-sovelluskehityksen pääosa-alueita ja kuvaa full-stack sovelluksen kehittämistä nykyisten teknologioiden avulla havainnollistaen eri osa-alueita käytännöllisemmin. Tutkittiin erilaisia internet-resursseja web-sovellusten kehittämisestä. Full-stack verkkosovellus kehitettiin, julkaistiin verkossa ja lähdekoodi tallennettiin julkiseen GitHubiin-arkistoon.

Tämän opinnäytetyön tärkeimmät tulokset olivat yhteenveto kaikista web-sovelluskehityksen pääalueista ja prosessikuvaus full-stack web-sovelluksen kehittämisestä. Tulokset osoittavat, että vaikka sovelluskehitys voidaan jakaa muutamaan pääalueeseen, on selvää, että jokainen alue vaatii syvällistä tietämystä, jota on myös jatkuvasti päivitettävä ja laajennettava tekniikan kehittyessä. Tämä paljastaa oppimisresurssien tarpeen, jotka myös kehittyvät jatkuvasti.

ASIASANAT:

Web-ohjelmointi, tietokannat, sovellus arkkitehtuuri, DevOps, full-stack

CONTENTS

LIST OF ABBREVIATIONS **6**

1 INTRODUCTION **6**

2 WEB APPLICATION DEVELOPMENT OVERVIEW **8**
2.1 Databases and backend 8
2.2 UI/UX design and frontend 12
2.3 Testing 14
2.4 Application architecture and development practices 15

3 DEVELOPING A FULL-STACK WEB APPLICATION **18**
3.1 Creating the Software Requirements Specification (SRS) Document and planning the work 21
3.2 Starting development and setting up the project online 22
3.3 Designing UX/UI with Figma and Tailwind UI 26
3.4 Developing the application features 28

4 CONCLUSION **31**

REFERENCES **33**

APPENDICES

Appendix 1. Software Requirements Specification (SRS) Document

FIGURES

Figure 1. Example of an ER-diagram (Peterson, Richard 2022). 9
Figure 2. An example of a 3-tier application architecture. 16
Figure 3. DevOps model describing some common practices that are included in the model (AWS). 17
Figure 4. A part of the Improvement application ER-diagram. 22

PICTURES

Picture 1. Tests for registration and authentication API endpoints. The tests start with test_ prefix and check_access_token_response is a helper function for making test assertions. 24
Picture 2. Example of how the Trello application looks. Cards include work that was done for the thesis. 27
Picture 3. Figma mock UIs. 28
Picture 4. Screenshot of the live Improvement app dashboard view with example projects. 29
Picture 5. Screenshot of the live Improvement app board view with columns and cards. 30

TABLES

Table 1. A table row describing a single epic. 21

LIST OF ABBREVIATIONS

API	Application Programming Interface
CD	Continuous Delivery
CI	Continuous Integration
CRUD	Create Read Update Delete, short that is often used to describe basic functionality related to data manipulation
CSS	Cascading Style Sheets
DBMS	Database Management System
E2E	End to End
ER	Entity Relationship, an ER diagram describes the relationships of different entities
GUI	Graphical User Interface
HTML	Hyper Text Markup Language
IDE	Integrated Development Environment, an application that for example enables editing source code, building executables and debugging
MVP	Minimum Viable Product, a simplified version of a product that allows for a faster release, validation and feedback
NoSQL	“Not only SQL”, allows for non-relational large volumes of changing data
ORM	Object-relational-mapping, an ORM can convert between objects in code and database tables
REST	Representational State Transfer, a REST API is simply an API that has been designed and built with a commonly accepted standard

SQL	Structured Query Language, a language for querying and manipulating data in a relational database
UI	User Interface, the views that the user sees and interacts with in an application
UX	User Experience, how the experience of using an application is for a user

1 INTRODUCTION

Building a web application could be described as something similar to building a house or an apartment complex. Developing a modern web application with various functionalities consists of several different areas of technical and design expertise being put together in a way where everything works together to deliver some kind of value to the users of the application. When dealing with such projects, it would be important to have at least a high-level understanding of what such projects include as it can require a high level of commitment from all the parties that are involved.

Often theses that discuss developing a full-stack application are focused on the specific technologies that are used to build some kind of specific application and reading them will not necessarily help someone who is new or outside of the field to understand the full picture. Yet applications can often be built from the ideas or needs from people outside the field of software development. For these stakeholders, it would be beneficial to bring clarity to the whole process of application development so that they can make better, more informed decisions and also understand the magnitude of such projects.

In a report by an international IT research firm Standish Group, it was reported that on average only around 30% of IT projects are successful in terms of being on budget, on time and on target (Standish Group 2015). The percentage is better for smaller projects but there is a steep drop when project size is moderate or larger. A better shared understanding of what is required when building applications can help with the communication between the application developers and other stakeholders. Communication between these groups is crucial to build trust, set the correct expectations for a project, turn the ideas of the stakeholders into a useful application and a successful project.

The purpose of this thesis is to provide a better understanding of the big picture of web application development, it can give clarity to those who are looking to get into the field and help to decide which areas might be of the most interest to them. It can also help someone looking to produce an application through a contractor. It is important that when buying software development, the key stakeholders in a project would have a better understanding of what they are buying.

The current chapter has introduced the need for clarifying what web application development consists of. Chapter 2 describes the different areas of web application development and what they contain on a high level to give a good understanding of the whole without diving too deep into the specific technologies that are currently being used. The more practical chapter 3 describes the development process of a Kanban style web application with some of the similar functionalities from other known applications such as Jira or Trello. This section discusses more about the specific technologies used and why they were used. The process included:

- planning the application’s user and data requirements
- building an API server using Python, FastAPI framework and PostgreSQL database
- making some simple designs for the frontend using Figma design tool and Tailwind UI kit
- building the frontend using React, Typescript, Redux and Tailwind CSS
- both backend and frontend code were setup with a CI pipeline using Github workflows that run the tests when new code is added into the project master reporitory
- setting up the project online and a CD pipeline to automatically update the production application when new code is merged

The final chapter 4 reflects on the findings of this thesis and highlights a need for developing constantly evolving learning materials for the field of web application development.

2 WEB APPLICATION DEVELOPMENT OVERVIEW

In the following sections we will look at the bigger parts of what building a web application requires. These are the big pieces that are in general always needed. To some extent these pieces can be similar to mobile, game or other kind of software development as well.

This thesis focuses mainly on the general technical aspects of building an application. It is worth mentioning that there are also other important aspects. As an example these could be things such as user interface and user experience design, clarifying the idea of the application from something abstract to a concrete idea, coming up with a plan for a minimum viable product (MVP) and planning use cases which are then used to define the technical needs for the application. While these different things might be only briefly touched, they are still an important part to keep in mind as it is very difficult to build anything without a clear idea of what is needed and how it should look and feel like.

Another topic that is too big to go into deeply in this thesis but needs a mention is application security. This is something that a developer needs to think about while developing the backend and frontend features. It is also a big part of the application infrastructure and monitoring.

2.1 Databases and backend

Any kind of popular applications tend to deal with storing, manipulating, displaying or in other ways dealing with data. Often in these cases the application needs to store data to be accessed for later use so it will require a database where the data can be stored. However there are various different use cases as well that aren't related to storing user data but instead for example storing data that is used to improving application performance or storing application logs for monitoring purposes.

A database is an organized collection of structured information, or data, typically stored electronically in a computer system. A database is usually controlled by a database management system (DBMS). Together, the data and the DBMS, along with the applications that are associated with them, are referred to as a database system, often shortened to just database. (Oracle Cloud Infrasctructure.)

There are many different kind of options for storing data depending on the use case of the application. Deciding on database needs is often one of the bigger decisions to make when building an application. It could be very difficult to change the underlying database systems after an application has already been in use for a while. This is why it is important to think about what kind of use cases the application has and what data the application is going to need. How should the data be structured? Will the data be clearly defined or is there going to be a need to store large amounts of unstructured data? Does the data have to be stored for a long time? Is there a need to combine or query the data in complex ways? These are just some examples of questions that need answers when making a decision.

There are many different database technologies and on a high level most options can be split into two categories which are SQL and NoSQL databases.

SQL based database technologies can require more careful planning beforehand. They're a good option when there are clear relationships between the different kind of data the application needs. The data in this kind of databases is usually organized in tables that contain multiple rows of data with columns that contain different types of data. This kind of data can be described with entity-relationship (ER) models that show how the data is related to each other as can be seen in Figure 1.

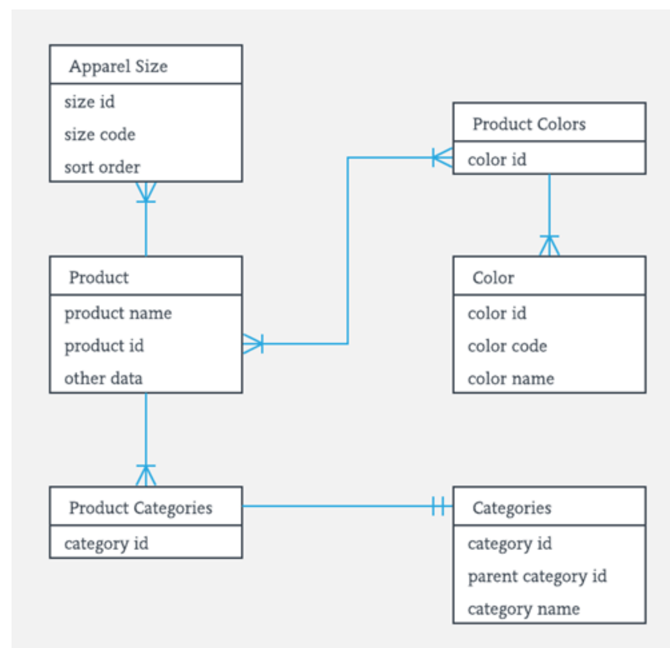


Figure 1. Example of an ER-diagram (Peterson, Richard 2022).

NoSQL based database technologies can allow for a faster start in development in a way that it isn't necessarily needed to have a clear idea of what kind of data and in what format or shape (data model) you want to store it. The format of the data can be much more simpler such as simply having a key-value pair where the key is unique and the value contains the data. When data is queried by the key, it returns everything that is stored as a value which could be data that is formatted in any way. This can be beneficial if in the beginning of development it is unclear what kind of data is required in the future or if you know that you will need to handle large amounts of data that possibly changes over time.

Specific database technologies are focused on different use cases. When the data has a clear model and can be organized into relations then a good option is to choose a relational database such as MySQL or PostgreSQL which are examples of SQL databases. If there's a need for less rigidity and constraints then a good option might be graph, document, key-value pair or wide-column based databases such as MongoDB which is a NoSQL document database or Redis that is a simple key-value pair NoSQL database. (Anderson, Benjamin – Nicholson, Brad 2021.)

There are also databases that have more specific use cases than just storing data. Such use cases are for example search and analytics of data, time series data and event streams data. Databases such as Elasticsearch or Opensearch enable search and analytics of large varied datasets. Databases like M3 or InfluxDB are time series databases that can be useful for storing for example application metrics and logs or other kinds of data that needs to be stored in pairs of time and value (Influxdata). Apache Kafka is an event stream platform that captures data in real-time from event sources like databases, sensors, mobile devices, cloud services, and software applications in the form of streams of events; storing these event streams durably for later retrieval; manipulating, processing, and reacting to the event streams in real-time as well as retrospectively; and routing the event streams to different destination technologies as needed (Apache Kafka).

When choosing between different database technologies, it also needs to be considered where the actual database with the chosen technology will be hosted. Will it be hosted on-premise which gives physical control over the hardware and software? Or will it be hosted on cloud or a third party provider? On-premise can provide more control but it might also be expensive as it requires handling all the hardware, handling licensing and having support personnel. Some technologies might be a proprietary solution that are

only available through a certain company. This binds the application quite tightly to that solution and if there ever rises a need to decouple, it might be a very difficult process. These are just some questions that also need to be considered.

When an application for example needs to store, fetch or update data and then possibly manipulate or format the data in different ways for displaying the data in a certain way in the application, these are all actions and logic that can be referred to as backend development. The data can either be accessed in a database or then through an application programming interface (API) that handles fetching the data from the database and provides it when requested. One way to think about it is that if a user interface (UI) connects a user to a device then an API connects pieces of software together.

An API defines a set of rules that describe how computers or applications can communicate between each other. An API can expose data and functionality in a specific way that allows others to use it easily through a documented interface. An API hides the implementation details of the functionality so it isn’t necessary for the user of an API to know how it works internally. This means a properly implemented API can change the internal implementation details drastically but it doesn’t matter for the user of the API as long as the way it is used and the data/functionality it provides stays the same. (IBM Cloud Education 2020.)

As an example one could build an API that serves the same data and functionality to a web application and a separate mobile application. This allows separation of concerns where the backend API handles all the business logic and data manipulation while the web and mobile application are separate applications only focused on representing the data and providing ways for the user to interact with the functionality provided by the API. This can allow easier scalability and also it can allow for different parties to use the same documented interface as well.

There’s a variety of different types of APIs, for example public APIs providing data that might not need any credentials to use or internal APIs that are only for a specific application that they have been built for (IBM Cloud Education 2020). One common ruleset for building an API is called REST which is a short for representational state transfer. It is a specific set of rules that define how applications or devices can connect to and communicate with each other (IBM Cloud Education 2021). When an API has been built following these standards it can be referred to as a RESTful API.

There are a variety of programming languages that can be chosen for backend development and when choosing the language there are a few things that should be considered. What language might fit the application requirements the best? What kind of talent is currently available on the market and how much can be spent on acquiring talent? How much time is available for the development work? Is it acceptable to rely on programming language frameworks that speed up development work or is there a need to control every aspect of the application by developing everything from scratch which can take more time and resources? How good is the documentation for the language? Often the language can be narrowed by answering these questions and sometimes compromises might be needed in one aspect or another.

2.2 UI/UX design and frontend

User interface (UI), user experience (UX) and frontend development are often tied together and while they can be thought of as parts of a bigger domain that focuses around what a user sees on an application and how they interact with it, the work these areas contain is very different. Usually if there is a bigger team building an application, each of these areas have their own specialist(s) working on it.

User interface design is the process designers use to build interfaces in software or computerized devices, focusing on looks or style. Designers aim to create interfaces which users find easy to use and pleasurable. UI design refers to graphical user interfaces and other forms—e.g., voice-controlled interfaces. In web and mobile application UI design often refers to specifically graphical user interfaces (GUI), meaning the visual controls on an application. There can however be voice-controlled interfaces (VUI) or gesture-based interfaces as well. (Interaction Design Foundation.)

User experience (UX) design is the process design teams use to create products that provide meaningful and relevant experiences to users. This involves the design of the entire process of acquiring and integrating the product, including aspects of branding, design, usability and function. While UI design is more concerned with the surface and overall feel of a design, a UX designer is concerned with the entire process of acquiring and integrating a product. (Interaction Design Foundation.)

While this thesis is more focused on the technical aspects of full-stack development, it is important to have some understanding of UI and UX design in the context of application

development. When planning development work, there is often a need for UI/UX designers to work together with the developers to get a shared understanding of what is needed from the user's perspective, what is technically feasible to do in backend/frontend and where compromises might be needed depending on available resources.

Frontend development is about building everything a user sees on an application and what needs to be built might be based on the work of the UI/UX designer(s). This consists for example of building the page layout, colors, buttons, sections, links, animations and in general all the different options for interacting with the application. The basic building blocks for these are HTML, CSS and the JavaScript programming language.

HTML is the basic block of web and it is a markup language that defines the meaning and structure of web content (MDN Web Docs). It could be thought of as the very basic blocks that are needed for building a web page. CSS is needed for styling the page. It is is a stylesheet language used to describe the presentation of a document written in HTML (MDN Web Docs). Finally building the interactivity on a web application requires JavaScript. It is a lightweight, interpreted, or just-in-time compiled programming language with first-class functions (MDN Web Docs). These three are the basics that are required for building web applications and have been staples for a long time. It should be mentioned that at the time of writing there is also a new type of code called WebAssembly that is a low-level assembly-like language with a compact binary format that runs with near-native performance and provides languages such as C/C++, C# and Rust with a compilation target so that they can run on the web (MDN Web Docs). However this is a new technology that is still at early stages (at the time of writing) while the previously mentioned basic blocks have been in use for around two decades.

While the basic building blocks for building web application frontends have been fairly consistent for some time, this is not the case with JavaScript frameworks. Programming frameworks help speed up development by providing easy to use APIs for doing common things that are often needed when building applications. They can abstract away a lot of code so there is less need for writing it and provide building blocks with good performance and good practices in mind. However there are a lot of JavaScript frontend frameworks, new ones come up very often and the existing ones can also change quickly. This can make it challenging to choose one and keeping up with the latest changes requires constant learning which can be tiresome for developers.

While JavaScript frameworks are not necessary for frontend development, they have grown in popularity for many years and the trend seems to be growing according to surveys such as Stack Overflow Developer Survey (Stack Overflow 2021) and the State of JavaScript survey (State of JavaScript survey 2021). The 2021 State of JavaScript survey conclusion notes that frameworks such as React and Vue have been dominant for 6 years so it seems that some frameworks have become more common than others. Again when choosing a frontend JavaScript framework for an application, questions need to be answered related to the available resources and what fits the application requirements – in a similar fashion as mentioned in the previous chapter when choosing a backend programming language. It should also be considered that since frameworks will very likely change over time and if the application uses a framework and it is expected to be running for a long time then it will also likely need steady updates.

2.3 Testing

Why would an application require testing? We write tests to be confident that our application will work when the user uses them (Kent C. Dodds 2019). It is essential to have good confidence in your application working as intended as it gets more and more features, grows in popularity and the application code grows in size. It would be very difficult to keep testing manually as new features are added and also making sure that when new things are added or functionality is changed that it doesn't accidently break something else. Having tests in place also gives more confidence when the code gets refactored.

Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior (Martin Fowler). Refactoring is something that happens almost daily when developing an application. Existing code gets improved in various small ways usually with a goal of either improving quality/performance, updating the code to use new technology or simply to make the code more readable for other developers. Since refactoring doesn't change the behaviour of the code and if there are tests in place, we can have better confidence that the application continues working as intended even after the refactoring.

What should be tested? Tests should avoid testing code implementation details and instead focus on covering the different use cases that the application has and they should also include cases for handling errors or so called unhappy paths. This means testing

the cases when the application is used correctly and also when it is used incorrectly or there is an unexpected error in the application.

Tests can in general be divided into four different types which are static, unit, integration and end to end tests. Static tests include things that catch typos and type errors as code is being written. Unit tests verify that individual isolated parts of the application code work as expected. Integration tests verify that several units of the application work together in harmony. End to end tests (E2E tests) include tests that click around the application and test all the different critical paths in the application at once in a similar manner as a real user would. (Kent C. Dodds 2021.)

Static tests should be very simple and easy to add. Unit and integration tests require a little bit more effort to add compared to static tests but they should still be fairly easy to add as most programming languages have frameworks that make it easy to write these kind of tests. E2E tests usually require more effort to setup and they can also possibly be more difficult to write and maintain. E2E tests also tend to take more time to run. Thus developers such as Martin Fowler and Kent C. Dodds recommend in general to have a base of static tests, biggest focus on unit and integration tests and then have a few e2e tests to cover some critical funtionalities (Martin Fowler 2019, Kent C. Dodds 2021). This kind of focus would essentially give the most bang for your buck in terms of time and resources spent on writing tests.

Lastly there's testing performed by real users. This kind of testing could be divided into two categories that are called hallway or corridor testing and user experience testing. Corridor testing refers to picking random individuals or for example a co-worker for testing the application. This is a quick and easy way to get feedback and improve quality. User experience testing would be specifically setting up a testing process for a more thorough testing by real users that are the application's target group. (Techopedia 2017, Järvenpää, Jarkko – Kovanen, Pasi.)

2.4 Application architecture and development practices

How can the previously discussed pieces of application development be put together to form the full application infrastructure and how does the development of such an application happen in practice? This chapter discusses designing the application architecture and how modern applications are developed and maintained.

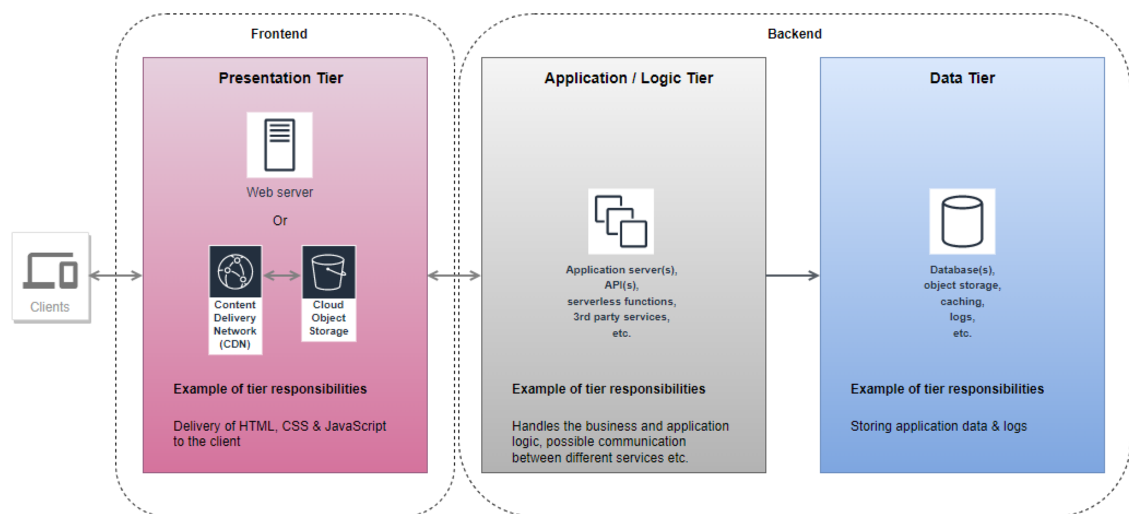


Figure 2. An example of a 3-tier application architecture.

The architecture of applications can vary widely depending on the use cases. It is also something that can possibly morph over time when for example the application becomes more popular and it needs to scale up to a larger user group all around the world. Figure 2 shows an example of a common three tier architecture of a web application. The figure shows some examples of what each tier might contain as an example but it should be noted that depending on the application, each tier might contain a large amount of different entities focused on specific things. A more detailed application architecture diagram might thus be much more complex. The main thing to note however is that each tier is concerned about specific main areas of the application (separation of concerns). This allows for easier scaling of the application.

When a complex application is developed and released for wide access, the work requires a host of different infrastructure and software to manage. This includes hosting the code in some manner for accessing the application. The example arcitechture shown in Figure 2 would require at least three instances for hosting the code (i.e. frontend code, application code and a database). Additionally the development and maintenance of the application requires storing the code (version control e.g. tools/services like Git and Github), systems for continuously updating the live application as code gets updated, monitoring the application for logs/analytics/security and possibly also systems for deploying different live environments of the application for only developer/testing purposes. For managing all of the previously mentioned things (infrastructure, monitoring, updates etc.), there is a commonly used model called DevOps.

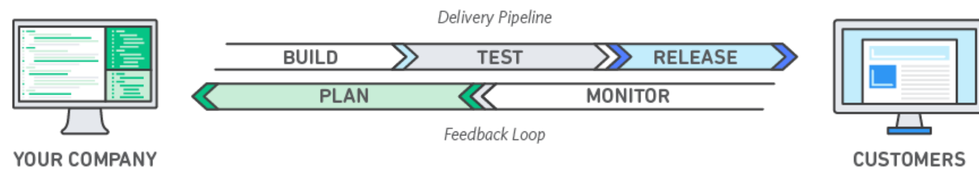


Figure 3. DevOps model describing some common practices that are included in the model (AWS).

Figure 3 shows the process of developing an application iteratively with the practice of DevOps. DevOps is the combination of cultural philosophies, practices, and tools that increases an organization's ability to deliver applications and services at high velocity. DevOps includes practices that aim to automate processes that might otherwise be manual and slow. Different tools are used to deploy code and provision infrastructure which in the past might've needed help from another team. (AWS.) In a way DevOps can be thought of as finding repetitive things or inefficiencies in a system that can be automated or otherwise made easier and faster. Some DevOps best practices include for example Continuous Integration (CI), Continuous Delivery (CD), Infrastructure as Code (IaC), monitoring and logging.

CI is a practice where changes to the application source code are merged regularly and tests are automatically run against the changes. This allows for finding possible issues faster and improves software quality. CD expands on CI in a way that if the previous checks in the CI pipeline have passed, then the CD pipeline will handle deploying the changes to a testing and/or a production environment. Together these practices are used to build an automated CI/CD pipeline that allows delivering incremental and reliable changes to an application very fast. IaC is a practice of using code to interact and configure infrastructure instead of doing it manually. For example system and hosting configurations can be automatically configured through code. Monitoring and logging are used to see how an application and related infrastructure is performing. This can be used to for example identify possible problems, possibilities for improvement and creating alerts if something unexpected happens. (AWS.)

3 DEVELOPING A FULL-STACK WEB APPLICATION

This section describes the process of building a full-stack Kanban style web application with some of the similar functionalities from other known applications such as Jira or Trello. This part goes into more details of specific technologies and ways of working that can be used. While the process of developing described here might not be fully comprehensive and they definitely aren't descriptive of every company's way of doing things, it attempts to cover the general steps that are most likely followed at some level currently in the majority of companies when developing an application. These can obviously vary a lot depending on the project size and on the available resources. In short the following sections describe:

1. The initial planning and creation of a requirement documentation for the application
2. The initial set up of the project for a fast development cycle
3. Design
4. The general flow of developing the application features

The main goal was to create a web application where the user will be able to register an account, sign in, create boards that can be thought of as projects, inside the boards the user can create columns and in the columns the user can create cards that can for example describe tasks that need to be done. The columns can then as an example describe the different states of the tasks and as the tasks progress forward, they are moved to different columns accordingly. There could be a lot of additional features to make this application more useful but this was the minimum viable product (MVP) goal for this application. The application consists of a REST API backend and a separate frontend React application.

The backend tech stack includes (not a full list):

- Python
- FastAPI framework – framework for building fast asynchronous Python APIs
- Pytest – for running tests
- PostgreSQL – a relational SQL database for storing application data
- Redis – an in-memory data structure store, used as a publish-subscribe (pubsub) messaging system

The FastAPI framework is a fairly lightweight framework to build performant APIs very quick. One of the advantages is that it automatically generates standards-based OpenAPI documentation for you. This documentation allows others to quickly see what your API provides and how to use it. Additionally it provides an OpenAPI specification file in JSON format that describes the API. This was useful as it allowed generating client code (code for making requests to the API) out of the spec file for use in the frontend React application. This allowed for the backend and frontend applications to play together nicely while also lessening the need to write a lot of code by hand.

PostgreSQL was chosen for the database mainly as a learning experience and also because the relations between the data in this application seemed fairly straight forward. The SQL query functions were written by hand and a python async database interface library called asyncpg was used for accessing and making the queries to the database. An ORM tool such as SQLAlchemy was also considered to speed up development but writing SQL by hand was chosen for a better learning experience. Writing your own SQL also allows for full control, possibility for fine tuning speed of the queries and it is one less dependency on a third party package.

The frontend stack includes (not a full list):

- Typescript – a “superset” of Javascript, adds static typing to the language
- React – one of the most popular frontend development frameworks for Javascript/Typescript
- Redux Toolkit / React Redux – Redux is one of the most known JavaScript state management libraries
- Tailwind CSS – a “utility-first” CSS framework for styling
- Tailwind UI kit – provides ready made professionally designed UI components built with Tailwind
- React Testing Library – for running tests on single components/screens
- Cypress – for bigger integration and E2E tests

Typescript requires using types for your variables (for example is the variable a string, a number or an object of a certain shape) and function return types, though through configuration you can adjust the requirements to be more or less strict. In small projects this might seem bothersome but when projects grow bigger, it becomes difficult and time consuming to navigate the code, understand the different variables and what they are supposed to contain. Typescript used together with an IDE provides great advantages

to help in development and lessens the chances of making errors in the code. In bigger projects this can lead to time saved because of less bugs in the code and better quality code. A type system works as a form of static tests and it could also be considered as a self documenting tool since the defined types describe what kind of data the application is handling. This can also be useful for other developers that join to work on the project. The other technologies used will be discussed more in the following sections.

There are some points worth mentioning about choosing third party packages developed by others. As a developer there are often many pros and cons to consider when selecting different technologies and packages to use. On one hand developers are often looking for speed or ease of development and some packages might offer this by for example reducing the amount of code and logic that you have to write. On the other hand relying on too many third party solutions might lead to problems down the line if for example your application relies heavily on some package and there is a bug or a security issue in the said package. It might take time for these issues to be fixed and you might have to make a temporary solution that is not ideal or perhaps the package you use isn't actively updated anymore and you need to switch to using something else. Another issue could be that your application uses tech that was previously under some free to use open source license and for some reason the license is changed to a more restrictive license.

The examples mentioned above are just a few among others that will have to be weighed when developing larger applications for commercial or general use. These considerations are also affected by how much resources are available for a project. A single freelancer coder or a small group of developers with limited resources will make different kinds of decisions than a large company. In this case the application was developed by the author alone. The guiding mindset for the selection of technologies was to avoid using too many packages but choose things that are currently quite popular and common to see in for example job requirements.

The application was named as Improvement. The source code is public and available in Github.

Source code for the Python REST API:

https://github.com/BadassHenkka/improvement-api

Source code for the frontend React application:

https://github.com/BadassHenkka/improvement

3.1 Creating the Software Requirements Specification (SRS) Document and planning the work

The project started by thinking about the user requirements and creating an SRS document for the application (Appendix 1). The user requirements were first written as epics. Epics describe some bigger piece of work that is broken down into smaller user stories. An example might be that a user wants to manage their project boards in the application. As seen in Table 1, this epic can include user stories such as the user wanting to create a new board. It can also include a story for editing or deleting a board.

Table 1. A table row describing a single epic.

USER EPIC ID	AS A	I WANT TO	USER STORY ID 1	USER STORY ID 2
1	As a user,	I want to manage my project boards	I want to create a new project board	I want to edit or delete a board

These stories together form the bigger functionality of the application and from these stories it was also easy to create the functional and non-functional requirements for the features in the SRS document.

These user stories and the requirements defined in the SRS document together formed the tasks that needed to be done to create the application. Planning and writing the requirements in this specific way can take some time but it takes away a lot of the guess work later on as it should be clear how the application should behave when writing the implementation code. The user stories can also help when writing tests for the application as they describe the way things should work from the user's perspective without going into detail of how the functionality is implemented.

As the plan was to first start with creating some parts of the application functionality in the REST API, the final step before starting the actual development work was to think about what kind of data the application will need based on the requirements and define some simple ER diagrams that describe the data tables and their relations in the PostgreSQL database. These diagrams can evolve over time as development of the application progresses but they help to visualize the different relations of the data. An example of a part of the Improvement application ER-diagram can be seen in Figure 4.

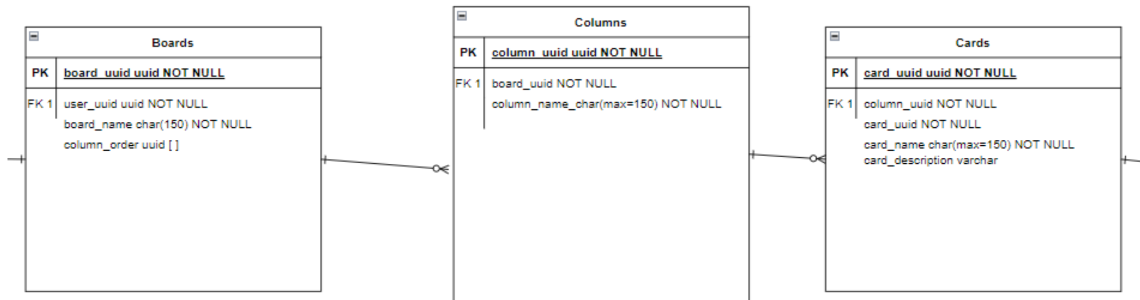


Figure 4. A part of the Improvement application ER-diagram.

3.2 Starting development and setting up the project online

The initial goal for the development work was to do a single application feature from start to finish, setup the backend and frontend code in Github with some CI/CD automation and also get the REST API and React application online. The idea here was to first build a base that would later on allow building full features one at a time by first building the backend portion of a feature, then the matching frontend functionality and to confirm it works well in a live environment as quickly as possible.

When developing a feature on your machine, a feature might seem to work great and fast but when it is deployed online, there might be unexpected issues or the application performs slower than expected due to latency introduced by for example database connections and chosen hosting providers. Developing a full feature, having it automatically tested and deployed online and being able to test it live quickly is very important for a fast feedback loop that allows for quick iteration and improvements.

The very first steps on the backend was to setup the project folder structure in a sensible way that separates concerns and helps navigating the code as the code base keeps growing. There’s a lot of ways to do this and there’s no one right way. Sometimes the programming language and possible frameworks that are used might dictate some specific way. One might also use a specific programming paradigm like object oriented programming or functional programming that might affect your decisions. However in general this just means trying to separate different parts of functionality into their own folders and files. As an example files including functionality that is related to REST API endpoints can be in one folder and the folder could contain files that are focused on specific endpoints providing specific functionality.

Additionally in the beginning a tool was setup called pre-commit. This tool allows setting up something called git hooks that can trigger actions when code is for example committed locally or when attempting to push to a remote repository. In this project it was used for automatic code formatting, checking the code style and types. This type of tool becomes handy when the project grows larger and you want to make sure that the code for example follows a specific style and good practices. It makes it easier to follow certain rules without having to think about them while coding and it is especially useful when a group of people are working on the same project.

The first application feature chosen for development was the possibility of a user to sign up and sign in to the application by typing in their username and password. The first steps were to write the backend parts. This work essentially consisted of creating the SQL, the endpoint with the functionality for registering a new user and an endpoint for getting an authentication token that the frontend application could store and use for authenticating API requests once a user is signed in to the application.

In this case it was fairly clear what needed to be done and a practice called test-driven development (TDD) was used for this initial work. This meant first writing tests that test the functionality that is needed before it even exists. This can create a nice workflow of thinking about what exactly is needed and then it makes it easier to focus on actually writing the implementation code.

```python
access_res_dict = {"accessToken": 1, "tokenType": 2}


def check_access_token_response(response, test_user):
    assert response.status_code == 200

    data = response.json()

    assert data.keys() >= access_res_dict.keys()

    token = data.get("accessToken")
    payload_sub = jwt.decode(token, settings.SECRET_KEY, algorithms=[settings.ALGORITHM]).get("sub")

    assert payload_sub.find(test_user.get("username")) != -1


def test_should_register_a_new_user_and_return_token(test_user, test_client: TestClient):
    response = test_client.post("/api/auth/register", data=test_user)
    check_access_token_response(response, test_user)


def test_register_should_fail_with_existing_username(test_user, test_client: TestClient):
    response = test_client.post("/api/auth/register", data=test_user)
    assert response.status_code == 400

    data = response.json()

    assert data.get("detail") == "This username is already taken."


def test_should_create_access_token(test_user, test_client: TestClient):
    response = test_client.post("/api/auth/access-token", data=test_user)
    check_access_token_response(response, test_user)
```

Picture 1. Tests for registration and authentication API endpoints. The tests start with test_ prefix and check_access_token_response is a helper function for making test assertions.

As can be seen in Picture 1, the test names describe what is expected, in this case they are very simple and we can’t say much about the implementation details other than that the endpoints should return a response with data that includes an access token that can be decoded or a message. It is a good practice to have tests that don’t test any specific implementation details that are used in the code. This allows for freely refactoring the code and the tests should pass as long as the expected end result stays the same.

After the initial version of the functionality in the backend was done, a Github Actions workflow file was created. Github Actions automate and execute software development workflows in your application repository. This allowed the creation of a workflow that

specified that everytime a pull request is opened in the project repository then the tests are run automatically. If everything works, the PR can be merged and that triggers another part of the workflow which is an automatic deployment to Heroku which is a cloud platform that allows running applications on virtual computers. They support a variety of programming languages and they provide a free tier for using their platform. Additionally they provide resources like PostgreSQL which was initially used as the application database. Later on during the development work, the database was moved over to a service called Aiven that provides fully managed open source data infrastructure.

After the backend portion of the work was done, some UX/UI mock ups were created of the sign up and sign in which is described more in the following section. The frontend part of the sign up and sign in functionality was then built and as it was the start of the project, that work also meant similar preparation work as with the backend. This included defining some basic folder structure for the application and adding some base configurations for linting/style with tools like ESLint and Prettier which help with keeping the code formatted nicely.

Again the practice of TDD was followed by first writing some simple tests for the functionality that was needed and then the actual components and functionality was coded. In this case it also worked nicely as there was a fairly good idea of what was needed and how it should be done. It should be noted that this is not always the case though. Quite often when working on something new, you might not have any experience of how to do something, it will require some research, trying things and possibly re-writing the initial solution even a few times after realizing there is a better way. In these cases it might be difficult to come up with sensible test cases beforehand if you don’t have a clear idea of how something should be done. However in these cases it can help to stop and think about what is needed. Writing down thoughts or even explaining them out loud can help to clarify things, that can help to bring out the solution and the tests can then be written afterwards.

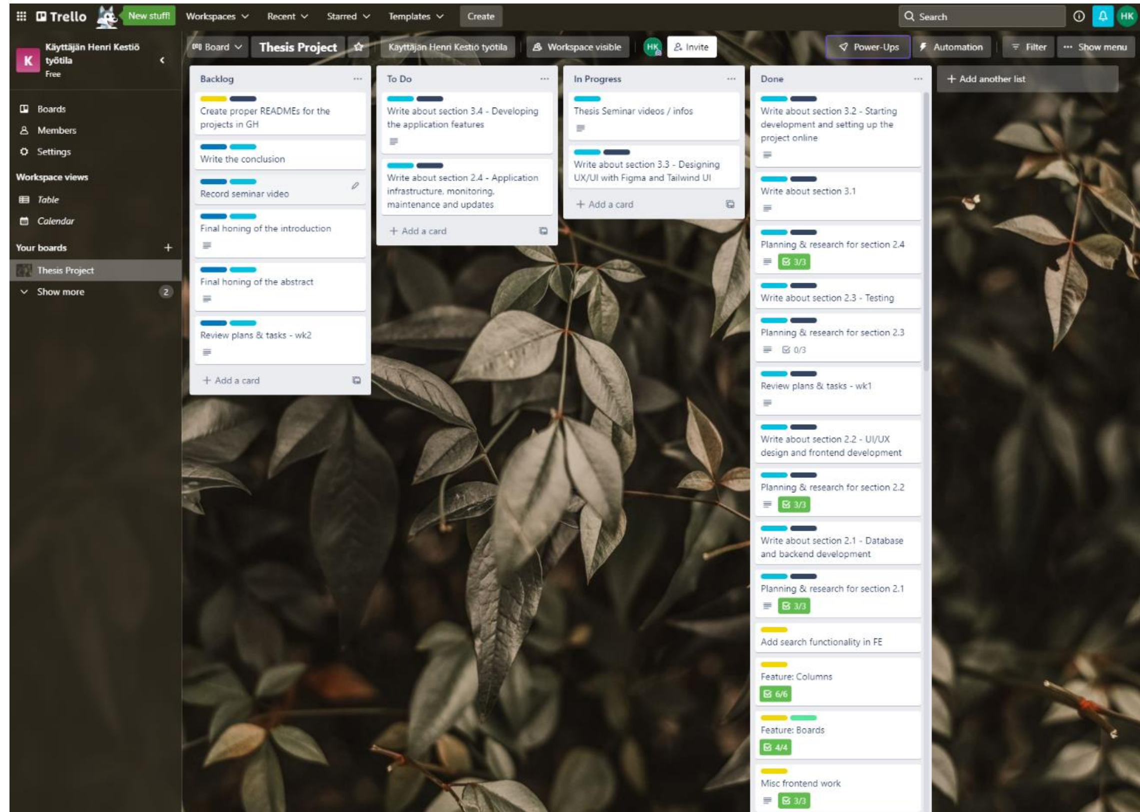
After the sign up / sign in features were done, a tool called husky was added that is similar to the pre-commit tool that was used in the backend project. In this case git hooks were added that automatically handle linting on commits and run the basic unit/integration tests (using libraries like jest and react testing library) and the browser end to end tests (using cypress framework) when pushing to Github. The frontend was then deployed to a service called Vercel that is specifically meant for hosting frontend applications that are built with different JavaScript frameworks like React, Vue or Angular

for example. Vercel offers a way to easily connect to the Github repository of the application, it creates a preview deployment on pull requests that are opened and automatically deploys a new version to production when the code in the main branch gets updated.

At this point the first full features of sign up / sign in were both done (initial versions) for the backend Python FastAPI application and for the frontend React application. Both applications were online (in "production") and both had some very basic CI/CD automation that ran tests and handled updating online production applications after the code was updated in the Github repositories. After this first initial work, adding new features was much easier as the base was built where adding new features would follow similar flow of first adding the backend functionality, then frontend UI parts and updates were handled automatically so it was easy to confirm how they work in live environment to get a fast feedback loop.

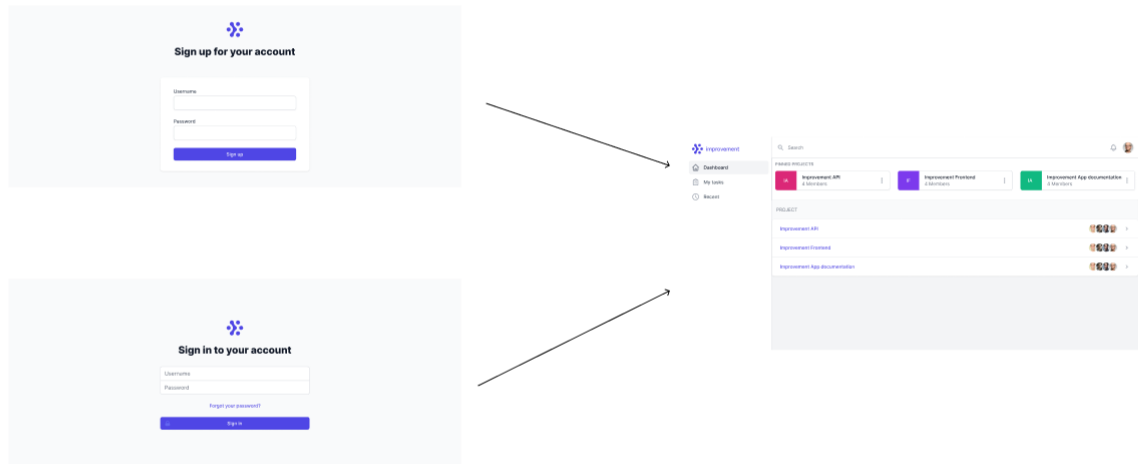
3.3 Designing UX/UI with Figma and Tailwind UI

The UX and UI design that was done in this case was pretty minimal since the idea was to mimic similar functionality and user experience from an application called Trello. An example view from the Trello application can be seen in Picture 2. However in reality this kind of work can take a considerable amount of time especially when creating something completely new. A part of design work can be to make ideas that are still fairly abstract into something more concrete and then find the parts that are especially relevant for a great user experience and make the application compelling. This kind of work can often happen after a SRS document has been created (or possibly during the creation) and before anything is even coded.



Picture 2. Example of how the Trello application looks. Cards include work that was done for the thesis.

For the Improvement application the work started by drawing some rough sketches on paper and writing down ideas. After this the ideas on paper were honed by creating some mock UIs in a tool called Figma. Figma is a web-based design and prototyping tool that also allows for collaborative work. As Tailwind CSS had already been chosen for styling in the planning phase, it was also decided to use a ready made UI package product called Tailwind UI. It provides ready made UI components for React that look great, are accessible and fully responsive for multiple different screen sizes. Additionally it provided Figma files containing the designs for the components. These could be then easily used to quickly create some mock ups such as shown in Picture 3.



Picture 3. Figma mock UIs.

In frontend development it is fairly common to use ready made UI components. These components can be used to piece together different pages and can for example come from third party libraries (e.g. for React there are libraries like Material UI or Chakra UI) or in some cases companies might have their own internal design system that contains internally developed components that have a specific style that follows company guidelines.

3.4 Developing the application features

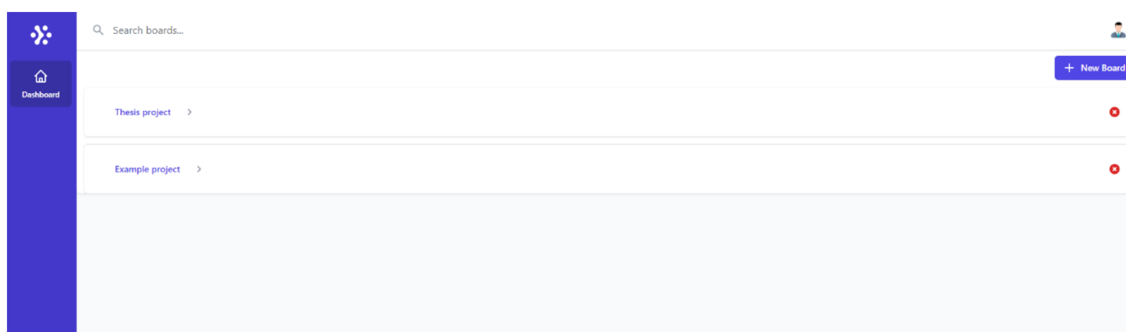
After the initial work was done, there was some refactoring done to improve some solutions and to clean up the initial work. Then it was time to start building rest of the functionality. As mentioned earlier, the MVP goals for the application functionality was to build the following:

- user sign up / sign in
- creation of boards that can represent for example a project
- in a board, a user can create columns with a name, change the column name, delete a column and change the column order horizontally
- inside the columns a user can add cards that for example describe a task and have the same functionality as with the columns except that additionally cards can be moved vertically inside a column and also they can be moved from one column to another

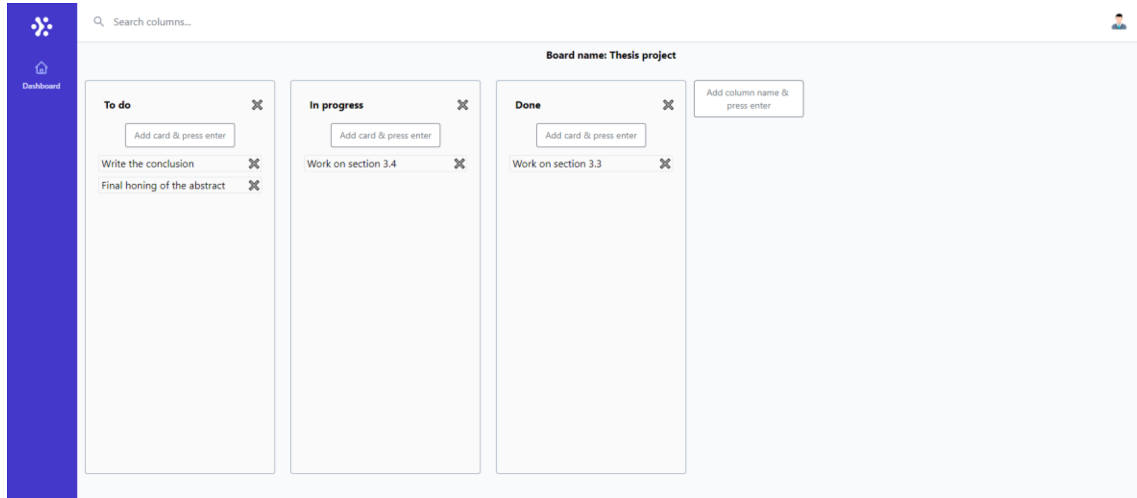
The plan was that the backend API would consist of HTTP endpoints and then a single websocket endpoint. The HTTP endpoints handle actions related to registration, authentication, fetching current user info and handling the board CRUD functionality. The websocket endpoint handles CRUD functions related to columns and cards. A user can edit columns and cards when they have navigated to a single board view. The reason for choosing to use websocket for these instead of HTTP was that it would better enable real-time updates when a user is on a board. The idea was that later on if a feature is added that enables multiple users to collaborate on a single project board, then everyone could see updates to the columns and cards in real-time. This collaboration feature was not part of the planned MVP goal for this thesis project but it would be built in a way that would allow adding it later on.

After the first feature of user sign up / sign in was done, it was fairly straight forward to continue adding new features and the workflow for a single feature generally consisted of the following:

- writing the SQL that adds a new database table and the required SQL functions that were needed for the particular feature
- writing the Python functions that call these database SQL functions to fetch the data and handle possible errors
- add a new API endpoint that uses the previously created functions
- add tests for the backend (if not created as a first step)
- create functions in frontend React app to use the new API endpoint
- create the UI components to add the required functionality and for displaying the data from the backend
- add tests for the frontend (if not created as a first step)



Picture 4. Screenshot of the live Improvement app dashboard view with example projects.



Picture 5. Screenshot of the live Improvement app board view with columns and cards.

Screenshots of the Improvement application at the MVP state can be seen in Picture 4 and Picture 5. Time spent on the development work wasn't recorded. Based on the commit history from Github, a rough estimate was made that the MVP goals were reached in around three months of work total. However it should be pointed out that commit history isn't great for measuring time spent, at times there were breaks of several weeks between the work, there were a few times that refactoring was done simply to try to learn new things and the estimate does not include creating the SRS document and designs.

What kind of future work could be done for this application? At this MVP stage, the feature set is very minimal and it would not gain any proper traction from real users. For example some key features from Trello that could be added are editable card names and descriptions, card labels, setting a time for finishing a task in a card, filtering/sorting of project boards/columns/cards, adding members to a board, real-time collaboration features and many more. Additionally there should be some features that would separate the application from Trello and other similar applications to make it a more compelling option for potential users. Lastly this MVP version didn't include any logging or monitoring of the application.

4 CONCLUSION

As mentioned in the beginning, the goals of this thesis were to explore the major areas of web application development on a higher level and give a general overview of what these contain. While each of the areas could have a thesis of its own and not everything could be covered exhaustively, the major areas of frontend development, backend development, testing and application architecture/infrastructure were covered that can then be used as a base for further studies depending on interests. Additionally the practical section shows how one might build a web application with some of the best practices in mind.

Even with this kind of high level overview of application development that should in theory stay fairly accurate for some time, it is still difficult to predict the future and to know if some parts and practices that were mentioned are out of date in just a few years. Constant change and learning is a part of software engineering. Further development based on this thesis could be carried out to better identify the concepts of each area of application development mentioned in this thesis (and more) and how they could be taught and studied in a way that also keeps evolving as technologies and practices do. The internet provides an immense amount of ever growing resources for this but it can be very difficult to filter out the necessary information for a level that a specific learner is at and combine that in a way that builds on top of their previous knowledge. Additionally, this kind of learning is not necessarily needed only for students that are in the field of software engineering. It might also be useful to have something similar for people who are outside of the field but who might still be involved in application development in some capacity such as decision-makers or clients buying development work.

From this overview, it is already clear that developing modern web applications requires a fairly large knowledge base that needs to be constantly updated and expanded on as technologies evolve and new ones emerge. New applications are constantly developed and they are more and more part of our everyday - not only for regular users but for example in business and organization settings as well. As the goal of applications is to provide some kind of value to their users, it means the development of applications usually also involve people outside the field of software engineering but who are experts on their own field that provide some insight to the application requirements. It is always better if the involved parties have a better shared understanding. Thus having an idea of

what the development of an application requires on high level such as described in this thesis can be very useful and additionally it is important for software engineers to be able to convey some of the technical aspects of development in an understandable manner. In this manner expectations are managed in a better way and trust can be built between the stakeholders. Any larger application project requires commitment from all the stakeholders and that will also continue after the application is initially ready as needs for new features arise and the application continues to be iterated on. As it can be with building and living in a house, an application is hardly ever fully quite ready either after the initial release. It needs some work and care afterwards as well.

REFERENCES

Anderson, Benjamin – Nicholson, Brad 2021: SQL vs. NoSQL Databases: What's the Difference? https://www.ibm.com/cloud/blog/sql-vs-nosql Referenced 19.2.2022.

Apache Kafka: Introduction. https://kafka.apache.org/intro Referenced 19.2.2022.

AWS. What is DevOps? https://aws.amazon.com/devops/what-is-devops/ Referenced 13.3.2022.

IBM Cloud Education 2020: Application Programming Interface (API) . https://www.ibm.com/cloud/learn/api Referenced 7.11.2021.

IBM Cloud Education 2021: REST APIs. https://www.ibm.com/cloud/learn/rest-apis Referenced 21.2.2022.

Influxdata: Time series database (TSDB) explained. https://www.influxdata.com/time-series-database/ Referenced 19.2.2022.

Interaction Design Foundation: User Experience (UX) Design https://www.interaction-design.org/literature/topics/ux-design Referenced 21.2.2022.

Interaction Design Foundation: User Interface Design https://www.interaction-design.org/literature/topics/ui-design Referenced 21.2.2022.

Järvenpää, Jarkko – Kovanen, Pasi: Software Development Buyer's Guide 2.0 https://vincit-com-media.s3-us-west-1.amazonaws.com/vincit-buyers-guide.pdf Referenced 28.2.2022.

Kent C. Dodds 2019: How to know what to test https://kentcdodds.com/blog/how-to-know-what-to-test Referenced 28.2.2022.

Kent C. Dodds 2021: Static vs Unit vs Integration vs E2E Testing for Frontend Apps https://kentcdodds.com/blog/static-vs-unit-vs-integration-vs-e2e-tests Referenced 28.2.2022.

Martin Fowler 2019: Software Testing Guide https://martinfowler.com/testing/ Referenced 28.2.2022.

Martin Fowler https://refactoring.com/ Referenced 28.2.2022.

MDN Web Docs: CSS: Cascading Style Sheets https://developer.mozilla.org/en-US/docs/Web/CSS Referenced 27.2.2022.

MDN Web Docs: HTML: HyperText Markup Language https://developer.mozilla.org/en-US/docs/Web/HTML Referenced 27.2.2022.

MDN Web Docs: JavaScript https://developer.mozilla.org/en-US/docs/Web/JavaScript Referenced 27.2.2022.

MDN Web Docs: WebAssembly https://developer.mozilla.org/en-US/docs/WebAssembly Referenced 27.2.2022.

Oracle Cloud Infrasctructure: What Is a Database? https://www.oracle.com/database/what-is-database/ Referenced 19.2.2022.

Peterson, Richard 2022: ER Diagram: Entity Relationship Diagram Model | DBMS Example https://www.guru99.com/er-diagram-tutorial-dbms.html Referenced 13.3.2022.

Stack Overflow 2021: 2021 Developer Survey https://insights.stackoverflow.com/survey/2021 Referenced 27.2.2022.

Standish Group 2015: CHAOS Report 2015. https://standishgroup.com/sample_research_files/CHAOSReport2015-Final.pdf Referenced 7.11.2021.

State of JavaScript survey 2021: State of JS 2021 https://2021.stateofjs.com/ Referenced 27.2.2022.

Techopedia 2017: Hallway Usability Testing https://www.techopedia.com/definition/30678/hallway-usability-testing Referenced 28.2.2022.

SOFWTARE REQUIREMENTS SPECIFICATION (SRS) DOCUMENT

Project name: Improvement Web Application

Date: 19.5.2021

Version: 1.0

By: Henri Kestiö

Revisions

Version	Primary Author(s)	Description of Version	Date Completed
1.0	Henri Kestiö	Initial requirements done	19.5.2021

Table of Contents

1. Introduction 4
2. General Description 5
3. Functional and Non-functional requirements.. 7
4. User and Software Interface Requirements 23
    4.1 User Interfaces 23
    4.2 Software Interfaces 23
5. Additional Software Requirements 25

1. Introduction

1.1 Purpose

The main goal of the project is to produce a web application named *Improvement* which is an application for creating Kanban style boards in which the user can create columns and cards that can for example describe tasks that need to be done. Once the project is finished, the application will be available online for users to register to and use freely.

1.2 Document conventions

Improvement — The name of the web application that is to be developed, stylized in italic

application — in this document, this will refer to the web application that is being developed during this project (named *Improvements*)

Kanban — a system for lean and just-in-time (JIT) manufacturing, the system takes it name from the cards that track production within a factory[1]

TDD — Test Driven Development, a method of writing tests first for a new feature, building the feature so that the tests pass and then possibly refactoring the code

BDD — Behavioral Driven Development, a sub method of TDD in where the tests focus on testing the behaviour of a feature/application and not the implementation details

CI — Continuous Integration, a system for automatically checking your code when new features are implemented, for example running code formatting checks and running the tests

CD — Continous Delivery, a system that automatically deploys code updates to the production site

UI – User Interface

UX – User Experience

1.3 Intended audience

This document is mainly for the sole developer of the application and it is intended to help clarifying the project scope and requirements.

1.4 References

[1] Description from https://en.wikipedia.org/wiki/Kanban

4.1 2. General Description

2.1 Product perspective

As mentioned in the project purpose, the product will be a web application named *Improvement* and it will take inspiration from other existing products such as Trello and try to improve on those.

2.2 Product features

The user will be able to register an account, create boards and inside them the user can create columns and cards that can for example describe tasks that need to be done. The user can invite other registered users to their created board, the users can assign themselves to a card and other board members can see who has been attached to a card.

2.3 User class and characteristics

The Ideal User – A user who has a project or a goal that needs to be broken down and categorized into several smaller tasks in a simple and visual way

Probable user examples based on occupation:

- Software Engineers
- Project Managers
- Coaches
- Athletes
- Teachers
- Students

Other possible user examples:

- High performers, people who in general are very goal oriented and want to organize their tasks

Regular people, people who have some big personal project or a goal they want to manage

2.4 Operating environment

The application itself will be used in a modern browser on a desktop or a mobile phone. This includes for example at least Chrome, Mozilla Firefox and Safari. Internet Explorer won't be supported.

The application's backend and frontend will most likely be hosted on Heroku. It's free to use for a set amount of hours in a month and it can also provide a Postgres database for the backend application. It also provides infrastructure for a CD pipeline which allows automatic building of the application code when new code is pushed to Heroku. There might be some other options to consider for the frontend but Heroku will be used as the starting point.

2.5 Constraints

There are no major limitations on the application implementation wise. Design wise, more careful thought will have to be put into the mobile user interface and user experience. Other than that is the free hosting services from Heroku which might be a bit slow sometimes. There are ways to improve it but this can impose some constraints when considering the performance requirements for the live application.

2.6 Assumptions and dependencies

On the backend API side, the application will rely heavily on the FastAPI framework which is currently about 3 years old and it itself relies a lot on Python's newer async functionality. While the framework is said to be production ready and it has some users from big companies (Microsoft, Uber, Netflix) it is hard to predict if some unforeseen issues might arise during development since it is fairly new. This is why the goal should be to try and get the API online as soon as possible so that it can prove its functionality and any problems that might surface can be fixed sooner.

Frontend will be done with React, Redux for state management and Tailwind CSS for styling. All are very well known and used frameworks/libraries that have been used a lot in production websites for a long time except for Tailwind. However the framework is very simple in the way it works so it is unlikely that there'll be any big issues with it.

CI/CD pipeline will rely on Github Actions and Heroku.

4.2 3. Functional and Non-functional requirements

3.1 Functional and non-functional requirements

The requirements here will be laid out based on user epics and user stories.

An epic describes a bigger functionality in the application and it can be broken down into smaller user stories. As an example one epic can be the application's user registration and authentication system. This is a single bigger functionality that is then broken down into smaller user stories that describe one small part of the bigger functionality. A user story is a simple statement that describes what a user wants to do in the application.

Functional requirements describe the user story in greater detail ie. what is the user's input and the expected result? Non-functional requirements can be related to things such as performance or looks ie. things that aren't related to the user's input or are perhaps more related to the implementation requirements.

Each epic (big functionality) contains a few different user stories. The epics are colored differently to make it easier to notice when the epic changes.

Epic 1 (authentication), Story 1

I want to sign up for the application easily.

Functional	Non-Functional
The user inputs their username and password (password twice), presses a button and it results in the user being automatically logged in and seeing the dashboard.	A successful sign up should log the user in to the app ideally in under 1s. This might be somewhat affected by the hosting platform of the application but 1 second should be more than doable.
If the user input's a username that is already taken, the sign up will fail and they'll get an error message informing them to use another username.	
If the given passwords don't match, the sign up will fail and they'll get an error message informing them to type the same password twice.	
The username will be required to have no spaces and it needs to have at least 3 letters.	

Epic 1, Story 2

I want to sign in to the application.

Functional	Non-Functional
The user inputs their username and password, presses a button and it results in the user being logged in and seeing the dashboard.	A successful sign in should log the user in to the app ideally in under 1s.
If the user inputs wrong credentials, they will get an error message informing them about it.	

Epic 1, Story 3

I want to be able to renew my password if I forget it.

Functional	Non-Functional
The user should be able to add their email in the application's user settings if they want to use this feature.	It should be clear that the email is required for this feature.
The user should be able to reset their password through a "Forgot my password" flow, assuming that the user has added their email in the settings.	The email should be sent out in less than 5 minutes.

Epic 2 (project boards), Story 1

I want to create a new project board

Functional	Non-Functional
The user clicks a button to create a new board, it opens a modal where the user can name the board and create it.	

Epic 2, Story 2

I want to edit or delete a board

Functional	Non-Functional
The user should be able to rename a board.	
The user should be able to delete a board after one warning prompt.	
The board should have some kind of settings where the user can do the above things and possibly other things such as adding/removing board members.	

Epic 2, Story 3

I want to use search to find a board

Functional	Non-Functional
The user should be able to search for their boards by name using a search bar input.	The search should quickly filter out boards that don't match the search.

Epic 2, Story 4

I want to sort my boards by the board creation time, last edited time or alphabetically.

Functional	Non-Functional
There should be a sort bar which indicates the possiblity for sorting by creation time, last edited time or sorting alphabetically.	The sorting should work near instantly.

Epic 3 (columns / task lists), Story 1

I want to create a new column in a board

Functional	Non-Functional
The user presses a button to create a new column, they can name it and then create it.	

Epic 3, Story 2

I want to edit or delete a column

Functional	Non-Functional
The user should be able to change the column name easily and possibly have some other options for editing the column.	
The user should have a way to delete existing columns where it gives them one warning prompt first.	If the column is empty, it can be deleted without any prompts.

Epic 3, Story 3

I want to reorder columns

Functional	Non-Functional
The user should be able to move a column sideways from one spot to another and immediately back if they want to.	This should be fast and smooth even with several columns (15+)

Epic 4 (cards / tasks), Story 1

I want to create a new task card

Functional	Non-Functional
The user presses a button to create a new card inside a column, they can name it and then create it.	The new tasks should appear quick inside the column even with several existing tasks (300+)

Epic 4, Story 2

I want to edit or delete a task card

Functional	Non-Functional
The user should be able to edit the card name, task description inside it, labels and other possible card settings.	
The user should be able to delete the card easily.	

Epic 4, Story 3

I want to use search to find a task in a board

Functional	Non-Functional
The user should be able to user search input to find the task card they are looking for by name.	

Epic 4, Story 4

I want to move tasks from one place to another and between different columns

Functional	Non-Functional
The user should be able to move a card sideways from one column to another and immediately back if they want to.	Moving the cards should happen quick and smooth even with several cards in the board (300+)
The user should be able to move cards up and down inside a column.	

Epic 5 (card labels), Story 1

I want to create a new card label

Functional	Non-Functional
The user should be able to create new colored + named (optional) card labels that can be attached to a card.	
The user should be able to pick the label color from a few existing colors or use a color picker tool.	
There should be a few (5) default labels that can be used.	

Epic 5, Story 2

I want to add card labels to cards

Functional	Non-Functional
The user should be able to add card labels to their cards on the board and the label color should be clearly displayed on the cards.	

Epic 5, Story 4

I want to edit and delete labels

Functional	Non-Functional
The user should be able to edit the existing labels – either color or name.	
The user should be able to delete the labels they've created but not the default labels.	

Epic 6 (project board members), Story 1

I want to invite other application users to my board by username or email

Functional	Non-Functional
The user can press a button and use some kind of input to invite other app users to join their board by username.	
The user should be able to invite others to their board by email if they're not already using the app.	The app should recognize if the email exists in the app's database and invite the correct user to the board, otherwise send an email.

Epic 6, Story 2

I want to remove myself from a board where I've been added

Functional	Non-Functional
A user should be able to remove themselves from a board easily.	

Epic 7 (card assignment), Story 1

I want to assign myself to a task card

Functional	Non-Functional
The user should be able to click a card and add themselves as a member of that card.	

Epic 7, Story 2

I want to remove myself from a task card

Functional	Non-Functional
If the user is attached to a card as a member, they should also be able to remove themselves from the card.	

Epic 7, Story 3

I want to assign someone else to a task card

Functional	Non-Functional
A user should be able to assign other members of the board to a task card.	

Epic 7, Story 4

I want to remove someone from a task card

Functional	Non-Functional
A user should be able to remove someone from a task card so that someone else can be assigned to it instead.	

Epic 7, Story 5

I want to quickly see who is attached to a task card

Functional	Non-Functional
All the members who have been attached to some task card should be displayed on the card clearly.	This should also work with multiple members (5+) attached to a card.

Epic 8 (card checkbox), Story 1

I want to create a checkbox list inside a task

Functional	Non-Functional
A user should be able to add some kind of checkbox list with a name inside a task card.	
The user should be able to add multiple checkbox items inside the list that are essentially smaller sub tasks of the container task.	

Epic 8, Story 2

I want to edit or remove the checkbox list inside a task

Functional	Non-Functional
A user should be able to rename the checkboxlist they've added to a task.	
A user should be able to remove the checkbox list easily if they want to. This would not have any warning prompts.	

Epic 8, Story 4

I want to reorder the tasks inside a checkbox list

Functional	Non-Functional
A user should be able to vertically reorder the task items inside a checkbox list.	

Epic 8, Story 5

I want to see a progression bar on the checkbox list

Functional	Non-Functional
The checkbox list should have some kind of progression bar or indicator attached to it which would show progression when the task items are marked as done.	

Epic 9 (card/task groups), Story 1

I want to select certain tasks to be grouped together

Functional	Non-Functional
A user should be able to create a task grouping and give it a name.	
The user could move tasks inside the task grouping or remove them from it.	

Epic 9, Story 2

I want to edit a task group

Functional	Non-Functional
A user should be able to rename the task group.	
A user should be able to add/remove members to a task group and they'd be displayed on the task group.	
A user should be able to add/remove labels to a task group.	

Epic 9, Story 3

I want to mark a task done inside a task group and see a progress bar for the task group

Functional	Non-Functional
A user can mark tasks “done” that are inside a task group and users should be able to see the progression of a task group.	

4.3 4. User and Software Interface Requirements

4.1 User Interfaces

UI and UX Design

TailwindUI - https://tailwindui.com/

- Provides good looking ready-made Figma design assets and React components that are responsive
- Tailwind CSS together with their UI kit provides a system to build good looking UIs faster while still allowing to also tweak them as much as needed

Figma - https://www.figma.com/

- Figma is a popular design tool that can be used on a browser
- Together with the TailwinUI kit, it'll be used to make the designs of the application user interface and plan out the user experience

4.2 Software Interfaces

Database

PostgreSQL - https://www.postgresql.org/

- An open-source relational database system that uses the tried and true SQL syntax
- Initially released in 1996, a popular database that is very reliable

Backend

FastAPI - https://fastapi.tiangolo.com/

- A python micro framework for building REST APIs fast which also includes auto generated API documentation
- Relies on Python's newer async functionality and can achieve very high performance that is on par with Nodejs APIs

Asyncpg - https://pypi.org/project/asyncpg/

- A python (async) database interface library for PostgreSQL, to be used to access and make queries to the database
- Based on the package developer's tests, it is on average 3x faster than the more commonly known psycopg2 (and its asyncio variant – aiopg)

Passlib - https://passlib.readthedocs.io/en/stable/

- Passlib will be used with argon2 hashing to hash & verify passwords that are stored to the database

Frontend

React - https://reactjs.org/

- Currently the most popular Javascript frontend framework

Redux Toolkit - https://redux-toolkit.js.org/

React Redux - https://react-redux.js.org/

- Probably the most known library for state management
- Has been around for a long time and has been proven to be reliable

Tailwind CSS - https://tailwindcss.com/

- A utility-first CSS framework that provides simple CSS utilities that can be used directly in the markup, builds into very small sized files and thus keeps the website's performance very high while still providing possibilities for building good looking websites with unique style

React Beautiful DnD - https://www.npmjs.com/package/react-beautiful-dnd

- This package will be used for creating the drag and drop functionality for the application's columns and cards
- A very popular, good looking and performant drag and drop library for React

4.4 5. Additional Software Requirements

5.1 Performance requirements

The application is intended to work and perform well with heavy daily use. The performance and speed of backend as well as the frontend is of great importance.

5.2 Security requirements

Data collection in the application will be kept to a minimum. The only thing that will be required from the user is a username and a password.

Username can be anything so it is not necessary to use a real name. If the user wants a possibility for resetting their password after forgetting it, then they can add their email in the application settings. If they do not wish to add it, then they will simply have to remember their password.

Even though data collection is kept to a minimum, security will be kept in mind when developing the application especially on the backend so that only the intended actions will pass. The application will also have SSL certificates to provide secure encrypted HTTPS connections.

5.3 Software quality attributes

The application will be developed using methods of TDD/BDD to produce a reliable application that works as intended. This means that when developing a feature for the application, a test will be written first that is going to test with the intended user input and the test will expect the result to be the inteded output. A test can also be written that expects a certain result when the user input is something that is wrong is not expected.

The tests will first fail as no code has been done. After this a solution will be coded to make the tests pass and afterwards it will be considered if the code can be refactored to be more efficient or reliable. Only then can the feature be considered done. The tests will focus on the user behaviour (thus the BDD method) meaning that the tests shouldn’t test the exact method of implementation. This means that if a feature’s implementation in code is changed afterwards, the tests should still pass as they don’t care about the way the expected result is produced.

A CI/CD pipeline will be setup to insure that any updates to the product will be tested before being updated to the production site. In essence, everytime updates are made to the code, a set of checks will run to first check the code style and then run the tests that have been written. This is the CI part. If these checks and tests pass, then an automatic CD system will run and update the code in the production site. At this point a staging phase will be left out.