TDDC88 Project - Company 1 - Architecture Notebook

Author: Company 1

September 2021

Contents

1 Purpose **2**

2 Architectural goals and philosophy **2**

3 Assumptions and dependencies **2**

4 Architecturally significant requirements **3**

5 Decisions, constraints, and justifications **3**

6 Key abstractions **6**
6.1 User interface 6
6.2 Application logic and back-end 6

7 Layers and architectural framework **7**

8 Architectural views **9**

Version	Author	Updates	Reviewed by	Date
0.1	Jacob Karlén	Initial version	-	2021-09-16
1.0	Jacob Karlén	Architectural overview diagram and architectural decisions	TBD	2021-09-17
1.1	Jacob Karlén	New decision and architectural philosophy	Daniel Ma	2021-11-02
2.0	Jacob Karlén	Added chapter 3, 4 and 6	Daniel Ma, Simon Boman	2021-11-02

1 Purpose

This document describes the philosophy, decisions, constraints, justifications, significant elements, and any other overarching aspects of the system that shape the design and implementation.

2 Architectural goals and philosophy

The goal of the architecture notebook is for it to be used as a reference and supporting document during the development of the application. The notebook should also give a good overview of the different aspects of the system architecture as well as describe significant decisions that have been made related to the architecture.

The architectural goals of the project is to find a technology stack and system architecture that supports the development of the patient journal web application, and that takes the requirements of the project into consideration. Since the focus of the project is on the front-end and user experience of the system, the back-end architecture doesn't need to be complex, and we can get away with a simple solution that delivers static data to the front-end.

The main limiting resource in the project is time, and this should be taken into consideration when deciding on the system architecture. This means that it could be motivated to use technologies that people in the team already have experience with, to speed up the process and reduce time needed to learn new technologies that takes away time from the actual development.

Another architectural goal is to try to limit third party dependencies in order to keep the project as stable as possible. Too many dependencies could affect the maintainability of the project in the longer term, with unexpected errors due to dependency changes.

In terms of philosophy, we strive to find a balance between a stable architecture that can easily be deployed into production, and at the same time being lightweight and nimble enough to support rapid development. We strive to avoid over-engineered solutions that make everything more complex than necessary, but at the same time we want to make use of relevant technologies that could be used in a real production environment.

Scalability is not of the highest importance since the application is only meant to be used by personnel of Region Östergötland (the customer). Reliability would however

3 Assumptions and dependencies

The system architecture will depend on a few assumptions and dependencies that are important to highlight.

The previous experience of web development and skill level varies among the developers within the company and this will most likely be a limiting factor of the project. Architectural decisions should take this assumption into account,

and prioritize technologies that at least some of the developers have experience with. Ease of getting started with the technology should also be taken into account.

The development will also be dependent on the data provided by Region Östergötland. Delays in delivery of test data from the customer will postpone the architectural work related to the back-end and data storage of the system.

Since Linköpings university will provide a Kubernetes cluster for deployment, the system architecture will be dependent on this in case it should be utilized. The project will be dependent on this in terms of server up-time and availability. The alternative of using a third party cloud platform like Heroku for deployment will lead to similar dependencies and since deploying to our own hardware is out of the question, this dependency will be accepted.

Other dependencies will include any third party libraries, frameworks and assets that we might decide to use as components of the architecture. These might in turn have dependencies of their own, which can quickly lead to a lot of dependencies in the application.

4 Architecturally significant requirements

The system architecture should support the implementation and fulfillment of the requirements in the Software Requirements Specification [add link to Gitlab]. Since the requirements need to be taken into account when deciding upon the system architecture, changes to the software requirements might lead to a need of updating the architecture in order to support the new requirements.

5 Decisions, constraints, and justifications

1. **Decision:** The web application will be built as a Single Page Application (SPA).
   Justification: Building the web application as an SPA comes with pros such as no need for page reloads when a user switches views. This will result in a faster and more interactive experience for the end-user, since only the necessary data will be requested from the server when interacting with the application. The web application will also be OS-neutral, something which is a technical requirement of the project.

2. **Decision:** Web Components will be used to modularize the UI components.
   Justification: We will use Web Components since it's a preference of the customer.

3. **Decision:** Angular will be used as a front-end framework for the web application.
   Justification: Angular seems to be the front-end framework that best fits

the customer's requirements and preferences. For starters it uses TypeScript and has native support for Web Components, something which our other main candidate, React, lacks. While it's still possible to use TypeScript and Web Components with React, it would most likely be more cumbersome. An additional pro of Angular is that there already exists plenty of great component libraries, for example Angular Material which is what we plan to use, as well as charting libraries like Chart.js to create good looking data visualizations (similar libraries compatible with React also exists). Another reason we decided to use Angular is the fact that a few people in the development team already had experience with the framework, so it is expected that we can be up and running faster. Angular is also more structured than React and uses the MVC (Model, View, Controller) design pattern which makes it possible to isolate the application logic from the UI layer. Angular also has standardized ways of creating components through the Angular CLI and is more structured than React, something which will be beneficial when working in a development team of around 10 people. Last but not least, Angular is also a complete framework which means that it comes with everything needed to handle routing, http-requests and testing, as opposed to React where you need to import third party libraries for these things. Not having to rely on third party modules for routing and other things makes the application more robust and maintainable over time (even though we of course will rely on the component libraries).

4. **Decision:** Node.js will be used with the Node.js web application framework Express for the back-end.
   Justification: Commonly used in combination with Angular and React so there are lots of great resources online for working with the chosen technologies. Node.js is basically server-side.js and it can (and will) be used with TypeScript in the project, which means that TypeScript will be used for both the front-end and back-end. Express is a web application framework for Node.js that comes with many great features for creating robust APIs. Since the the application will not be of such a large scale we could most likely have gotten away with most server setups, like a java or python server instead, but Node works very well with Angular since we then can use TypeScript on both the client and server. Node.js is also very scalable and fast so if there for some reason later would exist a need to scale up, the possibility will be there.

5. **Decision:** TypeScript will be used in both the front-end and back-end.
   Justification: TypeScript, which is a strongly typed subset of JavaScript, improves the readability of the code, since it makes everything more clear what type is to be expected. It also greatly improves debuggability because of this same reason. It will give you errors if you give a parameter of the wrong type to a function for example. It can help us spot errors in the code earlier in the project, and it is also nice to have language consistency

across the client and server, using the same syntax, and only having to switch the context we are working in.

6. **Decision:** TypeScript will be used in both the front-end and back-end. **Justification:** TypeScript which is a strongly typed superset of JavaScript improves readability of the code, since it makes everything more clear what type is to be expected. It also greatly improves debuggability because of this same reason, it will give you errors if you give a parameter of the wrong type to a function for example. It can help us spot errors in the code earlier in the project, and it is also nice to have language consistency across the client and server, using the same syntax, and only having to switch the context we are working in.

7. **Decision:** TypeScript will be used in both the front-end and back-end. **Justification:** TypeScript which is a strongly typed superset of JavaScript improves readability of the code, since it makes everything more clear what type is to be expected. It also greatly improves debuggability because of this same reason, it will give you errors if you give a parameter of the wrong type to a function for example. It can help us spot errors in the code earlier in the project, and it is also nice to have language consistency across the client and server, using the same syntax, and only having to switch the context we are working in.

8. **Decision:** MongoDB will be used as a document-based database for the application and Mongoose will be used for object modeling. **Justification:** This might seem like an odd choice for storing sensitive information like patient data, and that is most likely true. However, since information security is outside the scope of this project, and we will only be serving fake static test data through our back-end, we haven't taken security into account. We would probably not need a database at all, since we could serve the data from static documents right away, but we decided to use a MongoDB database either way, to make it at least a bit more realistic. MongoDB is often used in combination with Node.js+Express servers and it is easy to work with. We also have previous experience with MongoDB from before which will help. We will use the Node.js library, Mongoose, for object modeling of the MongoDB database.

9. **Decision:** The development environment will run in Docker containers orchestrated with Docker-compose. **Justification:** The motivation behind this is that it will create a consistent environment for everybody working with the development, so there shouldn't be issues related to the use of different dependencies on different systems. It is also convenient to be able to start the client, server and database in separate Docker containers with a single Docker-compose command. Another benefit of using Docker from the beginning of the project is making deployment with Kubernetes easier, since it is easy to go from using Docker-compose to Kubernetes.

10. **Decision:** MongoDB will no longer be used for storing data, and the data will instead be saved in a JSON-document on the server. **Justification:** The motivation behind this is that we came to the realization that MongoDB only added a layer of complexity to the architecture without providing new or useful functionality. Since we don't have any new data that needs to be stored and only work with static data, the database doesn't provide anything that we could not do with a static JSON document. When we removed MongoDB from the tech stack, it simplified the server considerably, since we don't need to write Mongoose schemas and queries for all the different data types.

6 Key abstractions

Here the main abstractions and critical concepts of the system will be described.

6.1 User interface

This section will list the important concepts related to the user interface of the system.

1. **Dashboard**: a UI page with a dashboard layout displaying information about a specific patient. The dashboard is in turn made up of several different components.

2. **Overview**: a UI page with a table displaying overview information about all the patients, and that can be used to navigate to the dashboard for a specific patient.

6.2 Application logic and back-end

This section will list the important concepts related to the application logic and back-end of the system.

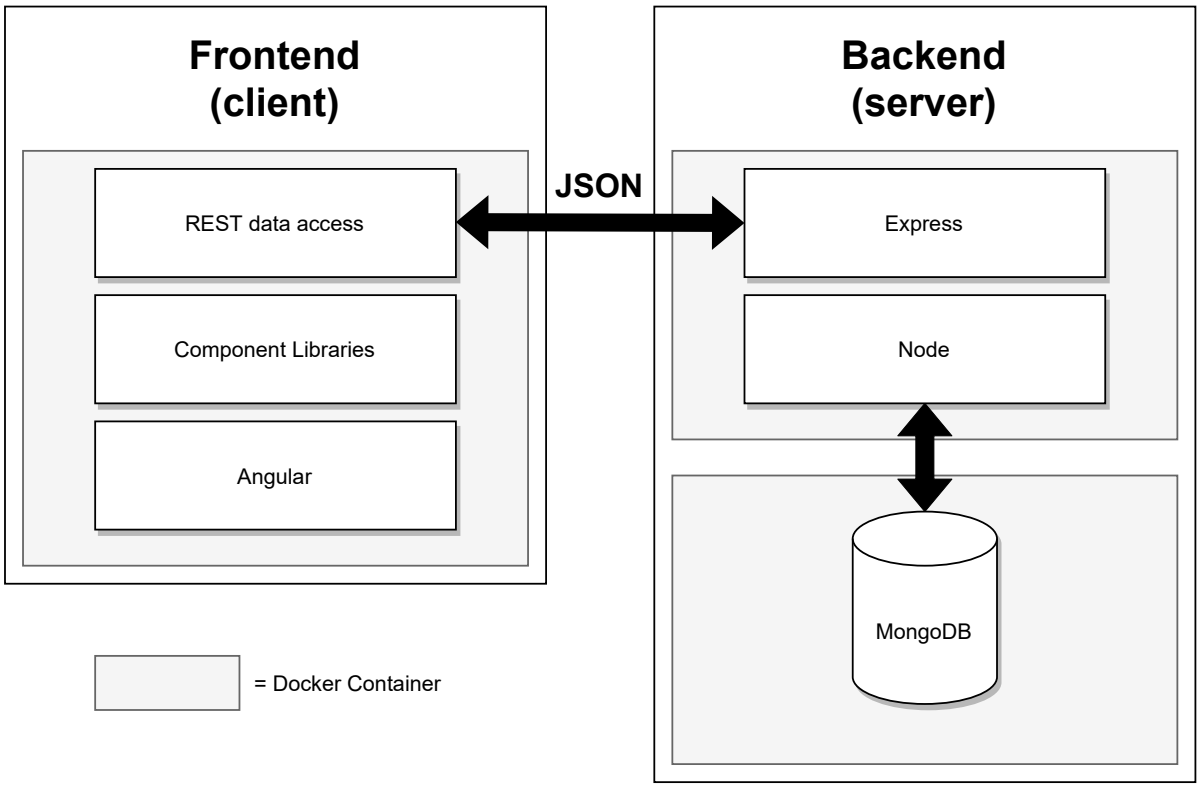
1. **Patient interface**: the interface defining a patient in the system. Defines the different attributes of a patient, such as patientId, name and vital parameters.

2. **Other interfaces**: more specific interfaces for modelling lab results, vital parameters, drugs, etc. that the Patient interface in turn make us of.

7 Layers and architectural framework

The architectural pattern that the system will be using is the client-server pattern, and this is a common pattern used in web development where the client mainly handles everything related to the user interface and then retrieves the actual data to be displayed from the server. The server will be thin and only provide routes for fetching patient data. Any additional computation and logic that needs to be done will be handled by the client, so it could be considered a fat-client. An overview of the system architecture can be found in the diagram down below.

Internal Scope



8 Architectural views