TDDC88 Project - Company 1 - Architecture Notebook

Author: Company 1

September 2021

Contents

Version	Author	Updates	Reviewed by	D
0.1	Jacob Karlén	Initial version	-	2021-
1.0	Jacob Karlén	Architectural overview diagram and architectural desicions	TBD	2021-
1.1	Jacob Karlén	New desicion and architectural philosophy	TBD	2021-

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 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 backend architecture doesn't need to be that complex, and we can get away with a simple solution that delivers static data to the frontend.

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 technolgies that take away time from the actual development.

Another architectural goal is to try to limit 3rd 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 dependecy 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.

3 Assumptions and dependencies

4 Architecturally significant 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 data needed 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 frontend framework for the web application.
   Justification: Angular seems to be the frontend 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 3rd party libraries for these things. Not having to rely on 3rd 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 backend.
   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 frontend and backend. 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 frontend and backend.
   Justification: TypeScript which is a strongly typed subset 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.

6. **Decision:** TypeScript will be used in both the frontend and backend. **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 frontend and backend. **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. Since information security is outside the scope of this project, and we will only be serving fake static test data through our backend, we haven't taken that fact 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 had previous experience with MongoDB from before which will help, and 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 use of different dependencies on different

systems. It is also really convenient to be able to start 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 that it will be easy when we want to deploy with Kubernetes, since it is easy to go from using docker-compose to Kubernetes.

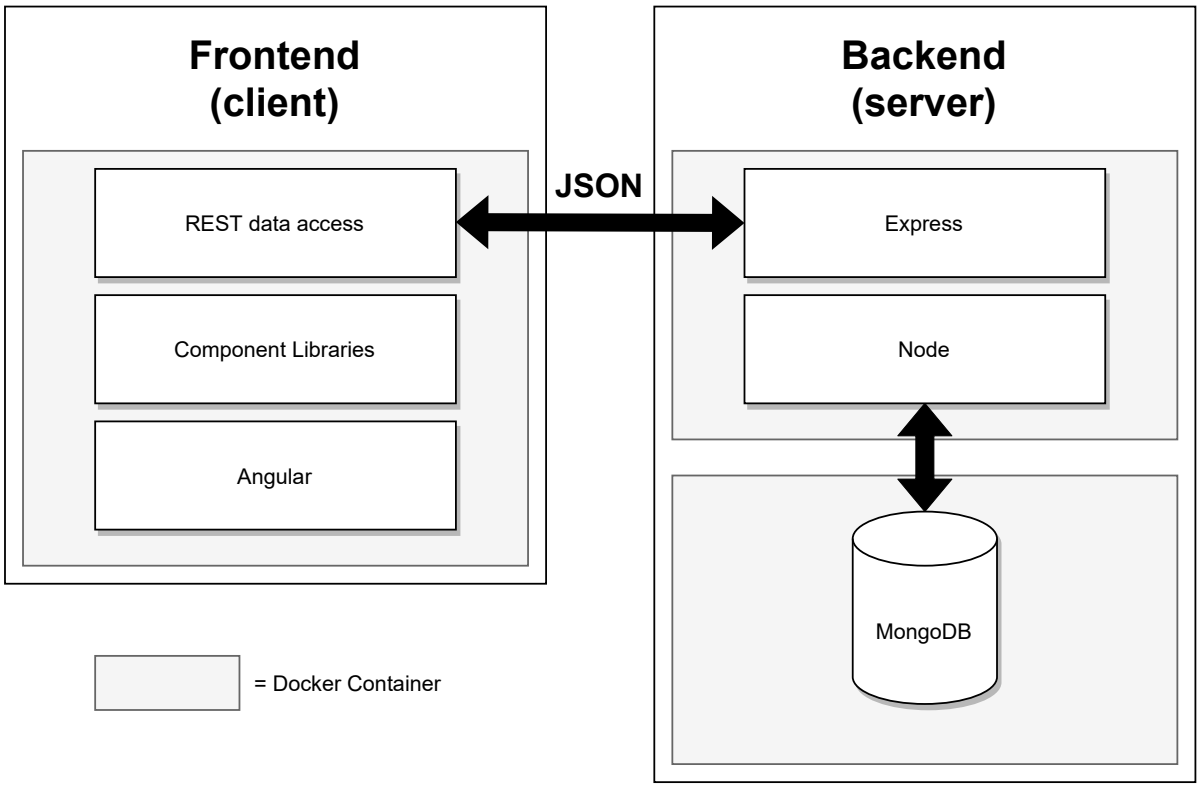
10. **Decision:** MongoDB will no longer be used for storing data, and the data will instead be saved in a JSON-document on the server **Justification:** The motivation behind this is that we came to the realization that MongoDB only added a layer of complexity to the architecture without providing new 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 right mongoose schemas and queries for all the different data types.

6 Architectural Mechanisms

7 Key abstractions

8 Layers and architectural framework

Internal Scope



9 Architectural views