TDDC88 Project - Company 1 - Architecture Notebook

Author: Company 1

September 2021

Contents

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
2.1	Somaye Gharedaghi	Grammatical Review	Jacob Karlén	2021-12-03
3.0	Jacob Karlén	Changes to chapter 2, 3, 7 and 8 based on feedback from supervisor and reviews.	Lucas Hultqvist	2021-12-04

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.

The system to be developed is a patient journal web application, with a focus on patient data visualization and UX. The system is composed of a back-end (server) which stores static mock patient data and provides routes for retrieving that data, as well as a front-end (web app in browser) which makes requests to the back-end and retrieves patient data that it then displays in a user interface and visualizes in different ways. The web application is supposed to be a proof-of-concept exploring ways of visualizing the patient data, and is not supposed to be production-ready at the end of the project.

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 and 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.

The main limiting resource in the project is time, and this should be taken into consideration when deciding on the system architecture, i.e. choosing technologies that the team has previous experience of.

Another architectural goal is to limit third-party dependencies to keep the project as stable as possible.

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 concurrently, we want to use relevant technologies that could be used in a real production environment.

There are no critical constraints related to scalability in the project since the system is only supposed to be used by personnel within Region Östergötland. Data security on the back-end side is also not a critical constraint since the customer would hook up their own back-end to the system's front-end if they were to use components of the system in the future.

2.1 Evaluation of architectural goals

When evaluating how well the goals have been fulfilled at the end of the project, the conclusion is that the goal of finding a suitable technology stack for supporting the development of the application has been fulfilled. One could however argue that the second goal related to the limiting resource of time has not been optimally fulfilled, since it has taken longer for the developer team to get comfortable with the technologies than were at first expected. In hindsight it might have been better to use a simpler frond-end framework, since many have found Angular to be quite difficult to learn.

Commenting on the adherence of the philosophy, one can mention that there hasn't been any issues related to the stability of the architecture. There have however been issues related to the deployment of the application, where in hindsight more effort should have been put in earlier in the project. The idea of being nimble and avoiding over-engineered solutions have partly been adhered to, where an example of this is that a decision was made to remove the MongoDB database to simplify the architecture and save time.

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 company's developers, which will most likely be a limiting factor of the project. Architectural decisions should consider this assumption and prioritize technologies that at least some of the developers have experience with. The 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 the delivery of test data from the customer will postpone the architectural work related to the back-end and data storage of the system.

Since Linköping university will provide a Kubernetes cluster for deployment, the system architecture will be dependent on this in case the cluster would be utilized. The project will be dependent on this in terms of server uptime 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 many 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. Since the requirements need to be taken into account when deciding upon the system architecture, changes to the software requirements might lead to updating the architecture to support the new requirements.

It is mainly the non-functional requirements that are architecturally significant, while the implementation of the requirements related to use cases could be supported by many different choices of architecture. All of the front-end components could just as well be created with React as the front-end framework and the server could run on Python or Java while still supporting all of the back-end related requirements.

The most architecturally significant requirements therefore include:

1. **NUC-013:** The application shall be adaptable for screens such as ipad mini, desktop and phone

2. **NUC-014:** OS-neutral web application that doesn't have to be installed

3. **NUC-016:** Server calls should be through open API:s, for example based on REST/HTTPS

NUC-014 defines that it is an OS-neutral web application that should be developed and therefore removes things like native app frameworks out of the question. NUC-013 also demands that the chosen front-end framework has support for creating responsive applications that can adapt to the user's screen size (most modern web app frameworks support this, so it was not really a limiting factor here). NUC-016 puts limits on the choice of server architecture.

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, which is a technical requirement of the project.

2. **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, which our other primary candidate, React, does not use natively. While it is still possible to use TypeScript with React, it would likely be more cumbersome. An additional pro of Angular is that there already exists plenty of great component libraries, for example, Angular Material, which we plan to use, and charting libraries like Chart.js to create stunning data visualizations (similar libraries compatible with React also exists). Another reason we decided to use Angular is 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, which will be beneficial when working in a development team of around ten 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 it is required to import third-party libraries for these things. Not relying 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). Some of the arguments here related to long-term maintainability might not be as important though, considering the project as more of a proof-of-concept. The main argument would be the previous experience of using Angular.

3. **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 many great online resources 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 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 for some reason later there would exist a need to scale up, the possibility will be there.

4. **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 about 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.

5. **Decision:** The mock patient data will be stored in a static JSON document on the server.
   Justification: Note: At the beginning of the project, the decision was made that MongoDB (w. mongoose) should be used as a database for storing the mock patient data. Since there was no need for updating the mock data and storing new data, the decision was made to remove MongoDB (and mongoose) from the tech stack and instead store the mock patient data in a static JSON document. This simplified the tech stack considerably and also removed the hours needed to write all of the mongoose schemas for the complex data structures.

   Using the JSON document for mock data storage works well in this case since this is just a proof-of-concept, and we do not have any new inputs to the system that need to be stored. If Region Östergötland were to continue with the project later, they would hook up their own back-end system to the front-end (client), so there is no real need to put time and

effort towards creating a robust solution for data storage. Most of the data that would be stored would also be classified as sensitive and should most likely be stored in a more secure manner that is outside the scope of this project.

6. **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 and server 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.

7. **Decision:** Kubernetes will be used to deploy the application
Justification: Kubernetes will be used to deploy the application and the main justification for this is that we have access to competence in how to use Kubernetes through our technical supervisor. Other alternatives like Heroku have been explored, but with the conclusion that it would be too cumbersome to attempt to deploy to Heroku without any previous experience or access to help for the platform.

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.
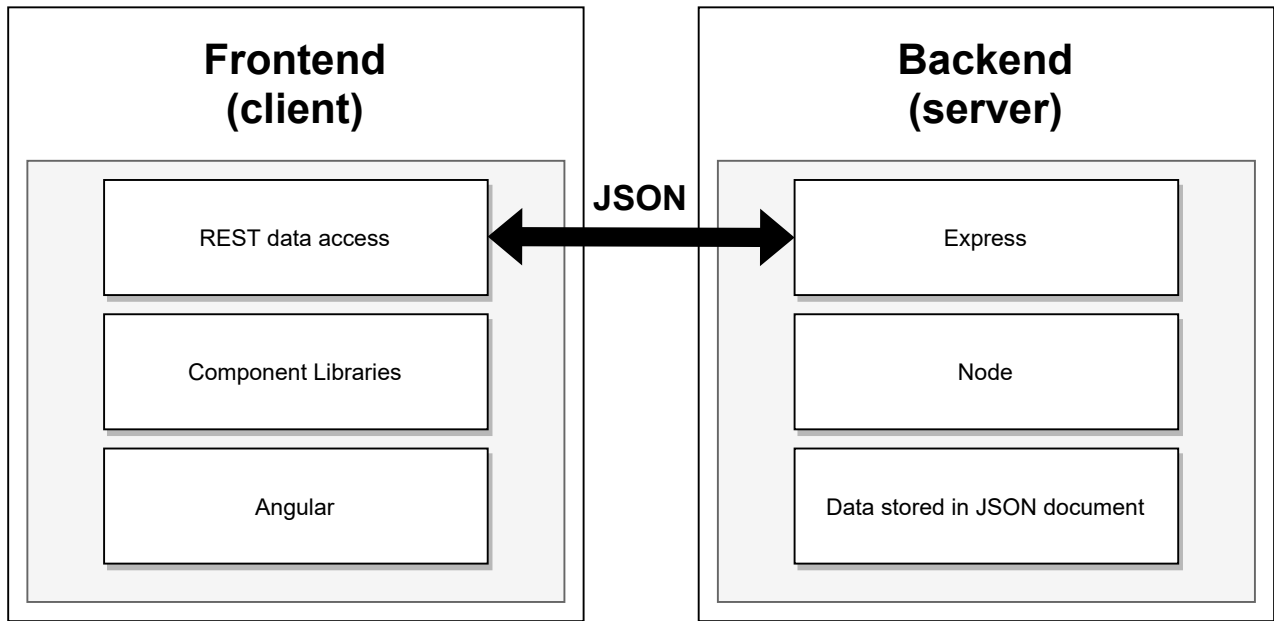
6.2 Application logic and back-end

This section will list the important concepts related to the application logic and back-end of the system.
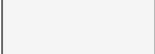
1. **Patient interface**: the interface defining a patient in the system. Defines the different attributes of a patient, such as patientId, name, and vital parameters.

2. **Other interfaces**: more specific interfaces for modeling 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. W

Internal Scope

Frontend (client)

REST data access

Component Libraries

Angular

JSON

Backend (server)

Express

Node

Data stored in JSON document

= Docker Container

8   Architectural views

8.1   Internal structure

In the simplified diagram below, an overview of the internal client file structure can be seen. The application itself is in its own folder *app* and all of the static assets like images are stored in the folder *assets*. The application code is split into three main parts *core, modules and shared*. *Core* includes the global UI components such as the *header* (navigation bar) and *footer*, as well as all of the Angular Services stored in the folder *services* that are used to retrieve data from the back-end. *Mocks* was used at the beginning of the project to encapsulate the code for delivering mock data that was stored on the client before the back-end and Angular services were functional.

The *modules* folder contains our Angular modules, and this is currently the two main modules *patient-dashboard* and *patients-overview*. Each of the module folders includes a *components* folder including all of the folders for the Angular components of the module and a *pages* folder including the main Angular components that represent actual pages to be displayed and acts as a wrapper for the other individual Angular components of the module.

The *shared* folder includes a folder for all the common data models of the application, and these are in the form of TypeScript files defining the different TypeScript interfaces. It also contains a folder for any Angular components that can be shared and reused across modules in the application.

The file structure for the server is considerably simpler and has all the TypeScript and JSON files in the same folder *src*. This simple structure of the internals of the server works because the server is very basic and only delivers static data through the Express routes defined in *routes.ts*. No calculations are done on the back-end, which simplifies everything quite a bit. The Angular patient service that is responsible for handling HTTP-requests to the server routes and retrieve patient data have service methods specified for each express route in the server. This makes it very easy for a developer to inject patient data into a component, since they just use the corresponding service method for the data they need (i.e vital parameters of a patient with a given patient ID).

The project is configured to use ES modules in both the front-end and back-end, with the "import" syntax rather than "require" (CommonJS).

```
client
├── app
│   ├── core
│   │   ├── footer
│   │   ├── header
│   │   ├── mocks
│   │   └── services
│   ├── modules
│   │   ├── patient-dashboard
│   │   │   ├── components
│   │   │   └── pages
│   │   └── patients-overview
│   └── shared
│       ├── models
│       └── components
└── assets

server
└── src
    ├── config.ts
    ├── index.ts
    ├── routes.ts
    ├── patient.ts
    └── patients.json
```