Architecture Notebook

Albin Kindstrand
Lucas Pettersson

2020-December-08

Name	Date	Change	Version
Lucas Pettersson, Albin Kindstrand	2020-12-08	Added better description of figure 7.2. Corrected small errors from document review 2: typos, bold text, removed boxed links when outputted.	1.0.
Lucas Pettersson, Albin Kindstrand	2020-12-07	Updated figures 7.1, 7.4, 7.5, 7.6 to latest version and added the new figures 7.2 and 7.7. Updated and added explanations to all figures.	0.9.
Lucas Pettersson, Albin Kindstrand	2020-12-06	Made many additions and changes to all parts of the document. A total revision of the document to prepare it for deadline. Updating i.e. significant requirements, architectural mechanisms, decisions. A lot of re-styling.	0.8.
Lucas Pettersson	2020-12-04	Added package diagrams (MVC and grouped). Added descriptions of all chapters.	0.7.
Lucas Pettersson	2020-12-01	Started making adjustments proposed in document review. Adding information about the value of the document to the customer. Fixing the order of change log.	0.6.
Lucas Pettersson	2020-11-17	Re-formatted the document in accordance to Company standards. Added this change log. Added introduction chapter, short chapter descriptions and more information about React and Recharts.	0.5.
Lucas Pettersson	2020-11-07	Formatted the document (tables, paragraphs, titles, sections).	0.4.
Albin Kindstrand, Lucas Pettersson	2020-11-06	Added architectural principles 1-6 and decisions 3-5. Created this LaTeX document for the architecture notebook and discontinued the Word document.	0.3.
Lucas Pettersson	2020-11-05	Added assumptions that drive architectural decisions A1-A5. Added decision 1 and 2.	0.2.
Lucas Pettersson	2020-11-04	Created document in Word. Added basic structure, purpose chapter and architectural goals (Maintainability, Simplicity, Reusability, Good integration between product and external services, Support, Security)	0.1.

Contents

Chapter 1

Introduction

1.1 Documenting architecture

The architecture notebook documents the software architecture of our product - Treet. It is a living document with the purpose of communicating the architectural decisions to all team members. The document also addresses the reasoning behind these decisions - from architectural principles and goals to assumptions, dependencies and relevant requirements.

1.2 Purpose

The purpose of the document is to communicate several aspects of the architecture. This is done to ensure there is a company-wide understanding for *how* the architecture is designed and *why*, to support development. This includes:

- Architectural goals and philosophies
- Assumptions and dependencies that drive decisions
- Architectural decisions, constraints and the justifications thereof
- Architectural mechanisms
- Architectural models and diagrams

1.2.1 Value of document after product delivery

Although the main purpose of the notebook has been to communicate the architecture during the span of the project, we argue that it is important for future reference. The document provides an insight into how the architecture came to be and can be of great value for further development. It is therefore recommended for Region Östergötland to continue working on the Architecture Notebook after the product delivery.

Chapter 2

Architectural goals and philosophy

The philosophical stance of the project is to build an architecturally lightweight system, connecting user and openEHR service via an intermediating application. The application is meant to extend the functionality of the openEHR service, providing patients with a user interface and functionality for interactive self-monitoring.

This chapter lists the architectural principles and goals which the designed system relies on.

2.1 Architectural principles

Architectural principles can be described as general rules or guidelines for the architectural resources and assets and how they will be handled. These principles lay the foundation for the architecture.

Principle 1	Information is an asset
Purpose	Information is an asset of critical value to I) the organization, II) it's users and III) the functionality and use of the application. Information is managed according to that. Information provides support for decision making: - For patients regarding e.g. when to report a measurement and to be able to view previous measurements - For health care professionals to make decisions based on the reported measurements By providing the correct information at the right time, users are able to make the right decisions.

Principle 2	Information accessibility and sharing
Purpose	Information is accessible to the users depending on role. Patients have access to the medical information they provide by self-monitoring. Correct information is provided at the right time to support decision making. Information is shared to authorized medical professionals.

Principle 3	Information security
Purpose	Information is protected from unauthorized access, use and alteration. Health and medical record information is protected by secrecy in accordance with Swedish law (Patientdatalag 2008:355). Health related data such as disease specific achievements or streak information that *can* be used to disclose a user's health status is also well-protected.

Principle 4	Reusable components or modules
Purpose	Components or modules should strive for being reusable, as a means of reducing overall development workload. Reusing components provides efficiency and consistency to the development.

Principle 5	Use already existing services where possible
Purpose	By using already existing services instead of developing the same functionality ourselves, development workload is reduced and the overall solution is improved. The customer already has services implemented in their organization (or in mind for future implementation) for parts of the system.

Principle 6	Use existing or similar external services to simulate a real implementation
Purpose	When using external services in the system, these services are the same or similar as the services already in-use by the organization. This is to replicate a real implementation of the system as much as possible. This will also easen the implementation of the system in a real environment, where the system can easily be connected to the services in use or the similar services can be switched out for the already in use ones.

Table 2.1: Architectural principles

2.2 Architectural goals

There are certain goals which the architecture must meet in structure and behaviour. These goals play an important part in the architectural design decisions made, as well as in future decision making.

No.	Architectural goal	Description
1	Maintainability	The architecture of the system aims to be easy to maintain, as per request from the customer. Easy meaning that it should take a smaller effort to repair or replace system parts or adapt the system to meet new requirements.
2	Simplicity	The architecture aims to be simple to facilitate easy development, building, deployment and maintenance. A simple architecture allows for clear communication to team members and a smaller development workload. This could also be a question of managing resources effectively.
3	Reusability	Aim to reuse modules to lower development effort
4	Good integration between product and external services	An architecture which is dependent on integrated services, must make sure these services are well-integrated.
5	Support	Support multiple smartphone devices and laying the groundwork to allow for further development for other devices such as desktop in the future.
6	Security	Information is protected from unauthorized access, use and alteration.
7	Extensibility	The architecture aims to have high extensibility to facilitate further development.

Table 2.2: Architectural goals

Chapter 3

Assumptions and dependencies

Assumptions and dependencies drive architectural decisions. In system design, we have to make assumptions on the behaviour and relationships of architectural elements before they are developed or implemented. Dependencies refer to the architectural elements our web application is dependent on for functioning correctly.

3.1 Assumptions

A1) Developing the solution as a web application will be stable enough and there is no need for developing a native application

A2) The processing of data (e.g. calculations) in the web application will be efficient enough to not put a high load on the web application, causing performance issues

A3) We can use openEHR to store medical data (Post/Get)

A4) We can use openEHR to store streak and achievement data as admin entries or text entries (Post/Get)

A5) Google Fit data can be integrated into the application via API

A6) Auth0 can be used for authenticating users

3.2 Dependencies

D1) The web application is dependent on openEHR for posting and getting data regarding streaks and achievements. This data is stored in openEHR, which makes this dependency critical for the app's functionality. However, the application is NOT dependent on openEHR for posting and getting *health data* in the current version of our product. This is yet to be developed before the product is deployed in a production environment.

D2) The web application is dependant on Auth0.com for authorization, no authentication functionality is provided by the application itself.

D3) The web application is dependent on Google's API:s to download data from Google Fit, however, it is NOT dependent on Google to manually add data directly in the solution.

Chapter 4

Architecturally significant requirements

Architecturally significant requirements is a subset of requirements, derived from the SRS, with the purpose of specifying which requirements have a measurable effect on the system architecture. These requirements can be functional and non-functional requirements as well as quality attributes.

In this chapter these requirements will be listed and their impact on the system design further explained or motivated.

Please note: In cases where multiple requirements have the same or very similar impact on the architecture, the first of these requirements has been brought up in this chapter. The related requirements (if any) are referenced in connection to the brought up requirement. All requirements are listed in the SRS. For the purpose of communicating the architecturally essential parts of each requirement, some requirements have alteration blocks ([...]) where information has been excluded. For the full requirement definition, please refer to the SRS. To see the full list of requirements, please refer to the SRS.

4.0.1 Functional Requirement 1

FR1: If a user enters a correct email and password combination [...] and presses the log-in button [...] the user shall be logged in and redirected to their start page.

Significance: This requirement pinpoints the need for, and intended use of, an authentication service. This relates to the Architectural principles *2) Information accessibility and sharing* and *3) Information security*. It also addresses goal *6) Security* in Architectural goals

Related requirements: FR2, FR3

4.0.2 Functional Requirement 5

FR5 : When a patient enters a new measurement, the system shall check for feasibility and display a message [...] asking the user to check their entry if deemed infeasible. The entry is deemed infeasible if the number is negative or contains any other characters than numbers.

Significance: This relates to the Architectural principles *1) Information is an asset*, describing that information has to be correct for the user's to be able to draw the correct conclusions based on the registered data.

Related requirement: FR6

4.0.3 Functional Requirement 7a

FR7a: Measurement graphs [...] shall display the progress of the patient's measurements.

Significance: This requirement assumes the implementation of functionality for displaying graphs from registered data. Programming graph components from the ground up can be technically challenging and time consuming given the knowledge level in the development team. To be able to fulfill this requirement efficiently, it would prove fruitful to use a library or external service to provide the mentioned graphs.

It relates to the Architectural principles *1) Information is an asset* and *5) Use already existing services where possible.*

Related requirements: FR7b, FR8, FR9

4.0.4 Functional Requirement 10

FR10: A patient shall be able to alter the degree the application is gamified [...].

Significance: This requirement addresses the need for a development pattern or library where the gamification level can be toggled between 3 states.

It relates to the Architectural principles *4) Reusable components or modules* and the Architectural goals *3) Reusability.*

Related requirement: FR4

4.0.5 Functional Requirement 11

FR11: A patient shall have the option to import data automatically from GoogleFit regarding the patient's physical activity, and have it registered to the corresponding measurement.

Significance: This requirement is significant because it addresses the need for integrating Google Fit into the application, and that the information provided by Google Fit shall be accessible for components handling measurements (updating or creating new). This relates to Architectural principles *1) Information is an asset, 5) Use already existing services where possible*, and Architectural goals *4) Good integration between product and external services.*

4.0.6 Functional Requirement 13

FR13: When a patient has completed an achievement, virtual collectables shall be rewarded to the patient, to be stored and viewed in their profile.

Significance: This requirement has to do with the storage of data, in this case achievement data. The architecture has to answer to how this data shall be stored. This relates to Architectural principles *3) Information security, 6) Use existing or similar external services to simulate a real implementation,* and Architectural goals *4) Good integration between product and external services* and *6) Security.*

4.0.7 Functional Requirement 14

FR14: A streak shall be displayed for the patient, representing how many days in a row that the patient has complied with their self-monitoring plan.

Significance: This requirement has the same significance as FR13 mentioned above, but regarding streak data.

Related requirement: FR15

4.0.8 Functional Requirement 20

FR20: A patient shall be able to set dependents. A dependent is set by entering the name and e-mail address of the dependent.

Significance: This requirement points out a potential security flaw, if not handled correctly. Since medical information is protected by secrecy, it is of high significance for the architecture to address this. This relates to the Architectural principles *2) Information accessibility and sharing* and *3) Information security* as well as Architectural goals *6) Security.*

Related requirement: FR21

4.0.9 Functional Requirement 24

FR24: A patient's virtual tree [...] shall grow when the patient's streak is being incremented.

Significance: This requirement is of great value to the customer and should be addressed in the architecture. The implementation of the trees is not the architecturally crucial part of this requirement, but rather the need for streak data to be stored somewhere for use in tree components.

Related requirements: FR26, FR28

Chapter 5

Decisions, constraints, and justifications

This chapter lists the decisions made to the architecture along with their justifications. By listing decisions and justifications, we communicate *what* has been decided and *why* regarding e.g. the use of certain techniques, libraries or frameworks or how different parts of the application are structured.

This chapter also lists the constraints put on the architecture, meaning what limitations there are to the architectural solution space.

5.1 Decisions

5.1.1 General decisions

Decision 1: Develop a web application instead of a native application

Justification: By developing a web application we facilitate both maintenance and usability. The trade-off is essentially between having an app that 1) is easy to maintain and available on all smartphone devices, but with more logical processing done in the web application or 2) limiting the available devices and putting more logical processing of data in the native app. The decision is based on the assumption that the web application will be stable enough and the processing of data will not be of great effort, hence there will not be significant performance issues that would be in favor for a native application.

Decision 2: Use Auth0 as authentication service provider

Justification: Auth0 is chosen to handle authentication requests to avoid the security issues and developer workload that may come from developing our own authentication module. Auth0 is a well-established and highly integratable service.

The Service was recommended by Region Östergötland due to the similarities with their ordinary platform, simplifying integration in their regular production environment in a later stage.

Decision 3: Mock data related to demographics

Justification: Limitations in the Better Care platform regarding the demographics module to handle data related to patient, organization and personnel forces us to create a mock service that provides our solution with the information necessary to work as intended.

Decision 4: Store game information in Better Care

Justification: By storing information related to the game elements in our solution as admin entries in the better care platform we can avoid using multiple back ends to handle information, thus minimizing dependencies to third party services as well as creating a more simplistic architecture.

Decision 5: Integrate with Google Fit

Justification: Integrating with Google Fit shows the concept of using already available data to enrich the health care with information as well as minimizing the effort needed by the patient to provide said information to the health care.

Decision 6: Mock data that does not add to the Proof of Concept

Justification: To be able to put as much effort as possible into showing our idea, the developing resources will be put into that area. This means down prioritizing efforts put into functionality that focuses on making the product commercially viable, but not adding to our proof of concept.

5.1.2 Decided programming languages, frameworks and libraries

---

Decision 7: Use HTML and CSS for UI structure and design

Justification: HTML and CSS are both well documented languages and the industry standard for creating, structuring and styling web pages and web applications. All modern browsers support HTML and CSS. There is a company-wide prior knowledge of HTML and CSS, meaning that self-education (if needed) will not be very time consuming.

Decision 8: Use Javascript for business logic and UI interactions

Justification: Javascript can manipulate the HTML DOM element and is designed to work with HTML and CSS. The development team (and other team members) have prior knowledge of Javascript or languages with similar syntax (i.e. Java). All modern browsers support Javascript.

Decision 9: Use ReactJS

Justification: ReactJS allows for building reusable components using HTML and Javascript, thus relating to decision 7 and 8. Specifically designed for building UI. Thouroughly documented. The prior knowledge of React in the developer team (and closely related teams) gives the project a quick start. This also facilitates education for other team members not familiar with React or web programming in general.

Decision 10: Use React-Bootstrap

Justification: Bootstrap is a well-established framework, providing the company with a well documented foundation as well as a solution RÖ can support in a later stage. The justification of using a library for component design is that it saves time (see Principle 5). This version of Bootstrap is built specifically for React and is one of the oldest React libraries, meaning that it has evolved over time. This is why React-Bootstrap is chosen over other, more general, HTML, CSS or JS libraries.

Decision 10: Use Recharts for creating line graphs

Justification: Customizable and lightweight charting library. Built on React components, allowing for quick and easy implementation in our product. Given the example uses of Recharts in the demo, it is clear that it can be modified to look as the prototyped line charts.

Decision 11: Use React Datepicker for creating date selection input

Justification: Locales for implementing calendar functionality. Customizable and lightweight library built on React components for quick and easy implementation in our product.

Decision 12: Use Base64 for encoding of API request

Justification: Module for decode and encode into Base64. Used to create the authentication part of the connection strings for openEHR. Needed for communicating with openEHR.

Decision 13: Use React Google Login

Justification: Module for implementing OAuth2-authentication while communicating with the Google API:s, in this case the Google Fit API:s. Google will not let us connect to their API:s without this authentication. React Google Login is an already built component, which saves time and hard work.

5.2 Constraints

No.	Constraint	Explanation
1	No e-mail integration	The LiU Kubernetes environment does not allow traffic over SMTP-ports, making communication with e-mail servers (or setting up our own) impossible.
2	No SMS integration	We have no access to any SMS-relay service making this type of integration non-feasible.
3	Web based application	The customer do want a web based application to fit the organizations current development standards and guidelines, therefore native apps for any platform are out of scope.
4	Storage of medical data	The customer require us to use the EHR platform provided by Better to store medical data. This is to make the solution compliant with reigning laws and guidelines such as GDPR and PDL.
5	Limitations to stored data	Limitations in what data can be stored in openEHR. Measurement, achievement and streak information can be stored, while demographics information can not.
6	Limited user privileges in better.care	Limitations in the user privileges granted to Region Östergötland by better.care limits the customer's, and by extension, our, solution space.

Table 5.1: Constraints

Chapter 6

Architectural Mechanisms

Architectural mechanisms detail recurring technical concepts or patterns in the application. These can be described as common solutions to common problems and together they form a solid ground for the software. The architectural mechanisms focuses on general areas - from application-wide integrations of services to development patterns.

M1: Login authorization

State: Implementation state

Description: Auth0 is decided on and implemented as the chosen authorization service

Related requirement: FR1

M2: Storing streak data and achivement data

State: Implementation state

Description: better.care is decided on and implemented as the chosen platform for storing data in openEHR.

Related requirements: FR13, FR14, FR24

M3: Components and views (React Workflow)

State: Implementation state

Description: Pages in the application are structured as views that import components, rather than all the functionality being in each respective view. This means that the components in a view can easily be switched out and if changes are made to the component, it affects the component in all the views where it is imported.

M4: Use React-Bootstrap cards in components or as the wrapper around a component in a view

State: Implementation state

Description: Using React-Bootstrap Cards library is a pattern developers have discovered to make the developed views look alike without having to make heavy adjustments to each view or component's CSS. A Card is a container with pre-made CSS and different attributes.

M5: Gamification level as props

State: Implementation state

Description: The user's gamification level is passed to the components and views as React props, making it easy to toggle what should be shown depending on the user's set level of gamification.

Related requirement: FR10

Chapter 7

Logical view

This chapter focuses on visually communicating the architecture with models. Each model is accompanied by a description clarifying it and adding additional information for ease of understanding.

7.0.1 High level architecture (HLA)

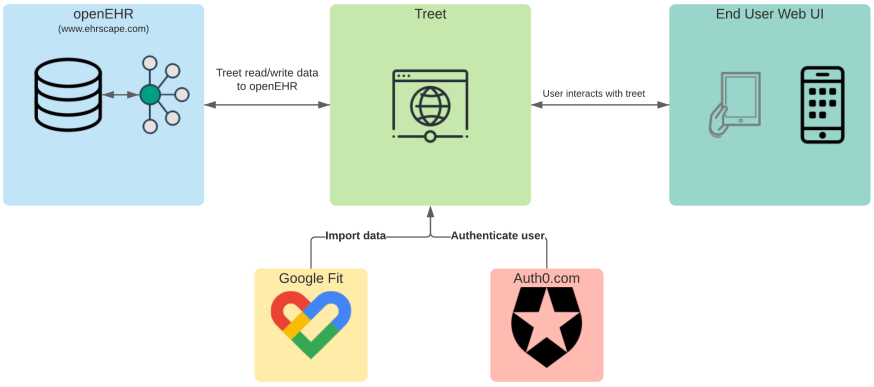

Figure 7.1: High level architecture v 3.0

Figure 7.1 is the high-level model of our architecture, showing the inter connectivity of the application in regards to external services (better.care and Auth0) and the external presentation layer (End user WEB UI). This model aims to portray, in simple terms, the fundamental parts of the system as a whole.

In Figure 7.1, the user connects to the application through the browser on their mobile device. When the user attempts to log in, the application

registers the user's interaction and initializes a connection to Auth0 for authentication. Auth0 returns whether the log in attempt has been successful or not. When the user is logged in and interacts with functionality using streak or achievement data, the application initializes a connection to better.care for getting the stored data, which better.care returns. If the user adds a new activity measurement in Treet, the user has the option to import data from Google Fit. If the user choose to do so, Treet initializes a connection to Google Fit, which returns the activity data specified.

7.0.2 Integration overview

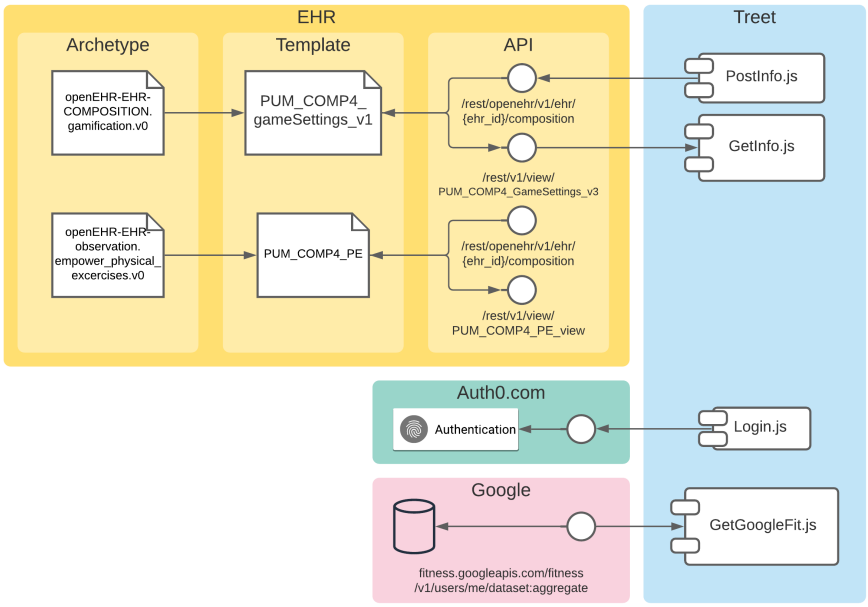

Figure 7.2: Integration overview

Figure 7.2 is an overview of internal components using third-party services and third-party services communicating with Treet. This model aims to portray dependencies as well as components providing the interfaces for integration.

The information exposed by the API:s builds on custom made templates derived from pre-existing and custom made archetypes. An archetype provides a place to define re-usable data and definitions, while templates consist of one or many archetypes or parts of one or many archetypes.

Please note: As visible in the model, no interface is developed in Treet to read or write medical data, although the templates and views are in place in

the ehr platform to enable further development. Please refer to Decision *6) Mock data that does not add to the Proof of Concept* in chapter 5.1.1 General decisions.

7.0.3 Client-server model

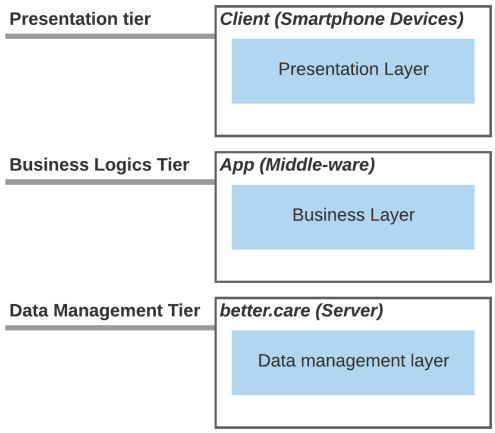

Figure 7.3: Client-server model of architecture v. 1.0

Figure 7.3 depicts the client-server model of our architecture. This model dissects the system into tiers: a presentation layer, a business logic layer and a data management layer.

The presentation layer is where the information is displayed for the user. The browser of the smartphone device is the client that displays what is being returned from the business logic layer - Treet in this case. The application acts as a middle-ware between the client and the server, handling functionality and calculations of data. The data management layer depicts the platform we use for storing data - better.care.

Example: The user adds their final measurement of the day by clicking "Save measurement" in their browser. The application registers the event and adds +1 to the Streak data. The application sends the new Streak data to the server via an HTTPS POST request. The application loads the view "Successfully added". The import of the streak component in the view "Successfully saved" triggers an HTTPS Get request to the server to get the updated streak data. The server returns the updated streak data and it is displayed in the application.

7.0.4 Package Diagram - Model-View-Controller

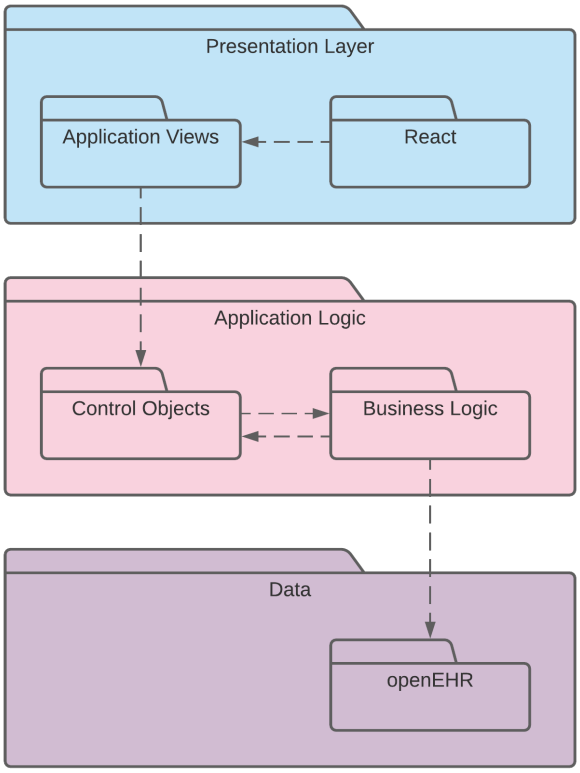

Figure 7.4: Package Diagram - Model-View-Controller v 1.0

Figure 7.4 depicts the architecture from a Model-View-Controller (MVC) perspective, separating the system parts into a presentation layer, an application layer and a data layer. This is done to show the general flow of communication between different parts of the system.

In Figure 7.4 the UI Framework React renders the Application View which is to be shown. The Application View is connected to Control Objects handling event triggers (i.e. mouse clicks), which are passed on to Business Logic. The Business Logic is made out of our components. The Business Logic communicates with openEHR in the Data Layer for getting and posting data.

7.0.5 Package Diagram - Groups of functionality

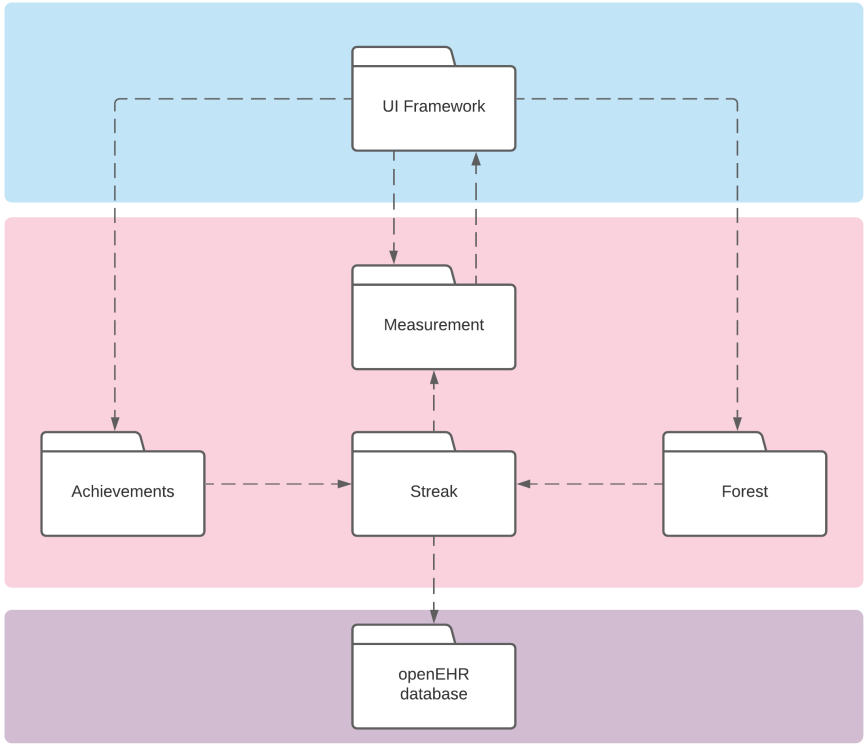

Figure 7.5: Package Diagram - Groups of functionality v 1.0

Figure 7.5 shows the main functionality groups of the application as packages. The dotted lines show dependencies between these groups of functionality.

UI Framework: The framework for rendering the application views and displaying the data retrieved from components.

Measurement: Measurement refers to the components and views related to creating or editing measurements.

Streak: Streak refers to the components related to calculating the streak based on registered measurements.

Achievements: Achievements refers to achievement related components and views (i.e. components calculating if the achievement have been completed and should be displayed, views that display the current achievment, the completed achivements and upcoming achivements).

Forest: Forest refers to forest related components and views (i.e. components calculating if a specific tree has been fully grown, views that display the current tree, the fully grown trees and upcoming trees).

openEHR: openEHR database refers to the server used to store data.

7.0.6 React Scope

Since React is not built out of classes, but instead of views and components, we have chosen to display what we call a React Scope (see Figure 7.6 below). This model shows the application levels from the React Framework perspective.

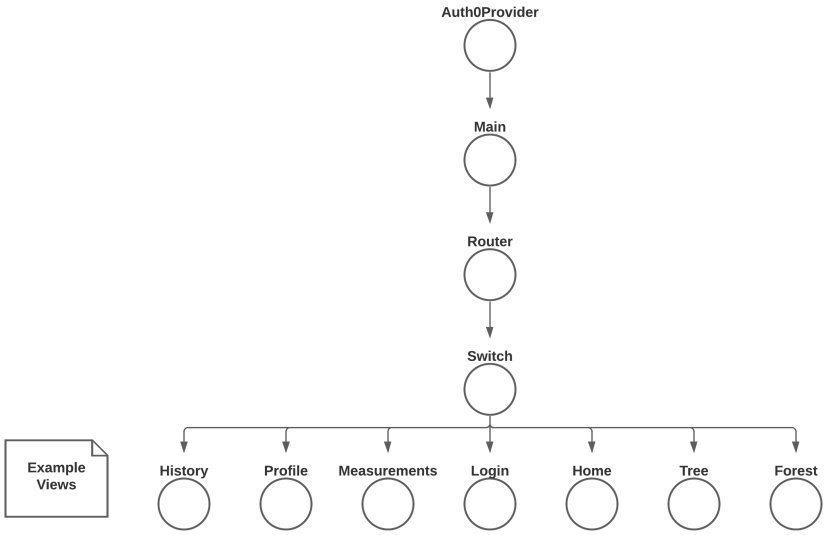

Figure 7.6: React Scope v 1.0

Auth0Provider is the initiator of the whole process. If the authentication with Auth0 is successful, Auth0Provider will send the user to Main. Main is the main view, which uses a Router and, by extension, a Switch, to toggle between the application views. The bottom level of the model shows different example views of the app.

Moving one level down, we find the components (see Figure 7.7). Different components are imported into different views, see chapter 6 Architectural Mechanisms, M3: Components and views (React Workflow).

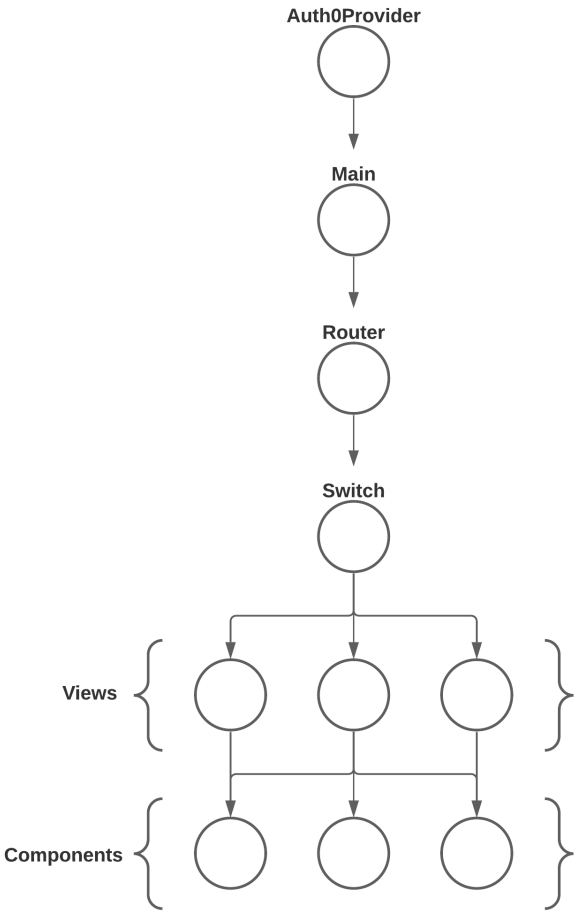

Figure 7.7: React Scope with components v 1.0

Chapter 8

Documentation for frameworks & libraries

8.1 Frameworks

ReactJS

ReactJS is the development framework used in Treet

- Overview https://reactjs.org/
- Getting Started https://reactjs.org/docs/

React-Bootstrap

Ready-made components from the framework/library React-Bootstrap are used to make development easier and more consistent

- Overview https://react-bootstrap.github.io/

8.2 Libraries

Base64

Base64 is used to create connection strings for openEHR

- Overview https://www.npmjs.com/package/base-64

React Datepicker

React Datepicker is used when a selected date span shall be applied to a list or graph

- Overview https://reactdatepicker.com/

Recharts

Recharts is used to create responsive line graphs

- Overview https://recharts.org/