Version: 1.2.4
Updated: 2020-12-07
Changelog: See last page.

Inspected By	Date
Cecilia Persson	2020-12-07

Architecture Notebook

Healthify

1. Purpose

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

2. Architectural Goals and Philosophy

The goal of the software architecture is to create a scalable and user-friendly web application intended for home monitoring of preadolescent patients with support for overviews for both parents and caregivers. The scalable aspect is referring to the scalability of the number of diseases that the web application can handle without having to rewrite much of the codebase.

The product will be finished and delivered to the customer, Region Östergötland, in the middle of December 2020. The product will however be continuously deployed, which means that Region Östergötland will have access to it during the development phase. During the development phase, the product will be developed continuously and might change a lot; the goal is to deliver a fully working product that Region Östergötland hopefully will find usable for their business. After the product is delivered, the company will have no responsibility for maintaining or continuously developing the product.

3. Assumptions and Dependencies

There are several assumptions and dependencies that set the scope for this project. The identified assumptions are listed below:

- Short timeframe: only about 3 months are given to produce the agreed-upon product for the client.
- Low to moderate experience of the entire company. This includes, but is not limited to: managers, developers, testers, analysts, and designers.

The identified dependencies are listed below:

- EHRScape provided by Region Östergötland.
- GitLab for enabling efficient source code version management and management of requirements.
- Flask will be used in order to build the web application and make the development easier.
- React is used for development of the user interfaces.
- PostgreSQL is the tool for the database development.
- Docker and Kubernetes will be used for enabling automatized distribution of the product to the customer.
- Continuous contact with customer for guidance and better understanding of what the customer really wants.
- The system will be developed under Apache License 2.0.

4. Architecturally Significant Requirements

Link to Software Requirements Specification: Press here.

- The system must be able to run on Google Chrome, Safari, Mozilla Firefox and Microsoft Edge and on various devices, such as a computer or a mobile phone.
- The system must use openEHR for patient record management.
- The system must handle persistence with a DBMS regarding account management.
- The system must be developed using CI/CD, for the benefit of the stakeholders.
- The system must be able to handle patient record input.

5. Decisions, Constraints, and Justifications

- React will be used as the framework since the app will run in multiple browsers. This decision is justified since React provides a component-based architecture approach which goes along with the short timeframe.
- React-Redux will be used since state management in React can be cumbersome.
- A DBMS will be used for handling accounts. However, this is only to showcase the product and will not be used in a real production environment. In a real production environment, accounts and credentials would be handled by a separate system such as BankID.

- A backend consisting of Flask will be used to provide a RESTful-API service which integrates with the DBMS. The choice of Flask is justified since the development team has the greatest experience with Flask compared to other backend frameworks. Furthermore, this backend will not be used in a real production environment, which lowers the requirements for it.
- Integration will be done with the RESTful-API for EHRScape provided by Region Östergötland, since OpenEHR is used by the customer today and would provide easy integration with current systems.
- GitLab CI/CD will be used since this gives many benefits for stakeholders, developers and the company itself.
- Kubernetes cluster will be used for CD, since this is a mature framework and is provided for free by the university.
- Swagger will be used for the Flask-backend, since it is an easy way for stakeholders and developers to explore the RESTful API. The Swagger-UI will always be available at:
  - http://tddc88-company-2-2020.kubernetes-public.it.liu.se/swagger-ui/.

5.1 Alternative Choices

- There are many alternatives when it comes to choosing a framework to work with. Some other frameworks we considered using were VueJS and Angular.
- The idea of having a simpler Python, Flask, and Jinja2 architecture is also being discussed. This was an idea as it would mean simpler implementations. It would also be easier since some developers have prior experience with Python, and it would reduce the complexity of the system.
- There were no alternatives to Git, as it is simply the most widely used and accepted version control system. When looking at a code hosting service, a popular alternative is GitHub. GitHub supports everything that GitLab does, but the main factor behind the choice GitLab was the university's GitLab
- account access.

6. Architectural Mechanisms

Persistence

To handle persistence regarding accounts, a DBMS will be used. The specified implementation method is PostgreSQL since it has a small footprint, and it will only be used during development. Persistence regarding patient information will be stored using openEHR, and the specified implementation method is EHRScape since it will be provided for free by Region Östergötland.

Usability

To be able to handle cross-platform usage we will use a frontend framework. The specified implementation method is React. Along with React, Healthify will also be working with Redux, that helps with state management in React. This choice was made as it would make it easier for all the developers to work with a store to help manage the state. This is especially useful as specific functionality in Healthify's application requires sending state information between different components and views and Redux helps in this aspect.

Efficient Development

In order to develop the software application efficiently and with less bugs, Healthify will use Kanban boards, Git, continuous integration, and continuous deployment. The specified implantation methods are GitLab CI/CD, Docker, and Kubernetes. See the documentation for the Process of Development for further information.

In GitLab, the issue and board features are used to create Healthify's own style of a Kanban board. Currently, the different boards include:

- Open: Here are issues that either are not validated, have some larger problem, are outdated, or yet not dealt with.
- Developers Validate: issues are placed here by analysts that have created them from requirements. The issue contains one or more requirements stated in the description as task(s). Developers validate the issue together to evaluate if it developable or not.
- Testers Validate: After developers have validated the issues to make sure that they fit the product and are not too complex, testers will also validate all the issues to ensure they are testable and relevant.

- Define interface: The UX designers, in collaboration with the developers and testers, decide the design of the features.
- Backlog: The issues that are ready to be developed and have been validated by both developers and testers and given UX-design (if needed for the specific issue).
- Development: Issues that are chosen for the current sprint and are assigned to specific developers.
- Reviewing: After implementation of an issue, they are moved into Reviewing; here, the code can be reviewed by peer review in groups of developers, or a manager can review them.
- Testing: After tasks are implemented and reviewed, they are ready to be tested by the test team.
- Not Accepted: If the issue is not accepted by the developers in the Reviewing stage or don't passes the tests in Testing, it is moved to Not Accepted and the developers need to review the issue.
- Closed: After testing is done and all tests have passed, the issue is closed. Otherwise, if tests fail, testers should give feedback on what failed and move the task back into Not Accepted.

The workflow for developers using Git involves working in feature branches and then merging the completed feature into the master branch. This makes it easy for developers to work on many different features at the same time, while keeping the master branch clean. The branches are merged by submitting a merge request: these are automatically run through the GitLab pipeline to make sure all features pass the tests and the developers also review the code, so it is understandable and correct.

Continuous integration and deployment are also done through GitLab. GitLab provides a nice interface to monitor all the stages in the pipeline defined in the continuous integration .yaml file. This file describes the stages and the jobs to be executed every time a change is pushed to the repository. These processes are described in further detail under Deployment Process in section 7.

7. Key Abstractions

Deployment Process

Figure 1 – Deployment Process below describes the overall deployment process. The master branch of the Git repo will always be a deployment ready branch, which will be continuously deployed via GitLab CI/CD and Kubernetes. A feature branch workflow has been chosen to easily be able to integrate and develop new features which are not deployment ready. When a feature is considered done, it can be merged into the master branch, which will automatically deploy it to the Kubernetes cluster.

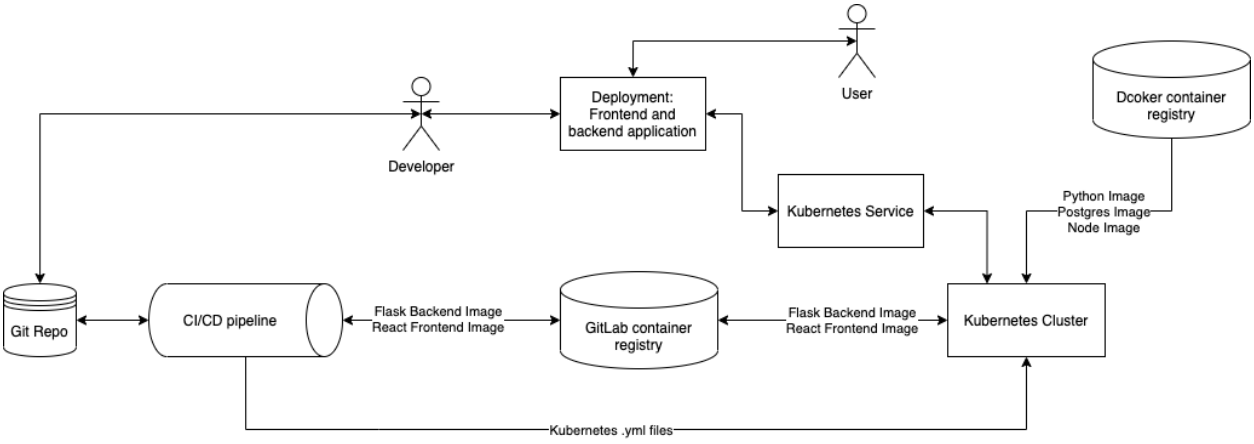

Figure 1 – Deployment Process

Central Datatypes

Figure 2 - Central Classes (below) highlights the three main classes implemented in the PostgreSQL database. This is to provide a way to handle authentication and persistence during the development phase of this project. If this application was to be implemented, this part would be handled in another way. The Child and Parent classes inherit from the user class through polymorphic identity in the database. This is done to more easily handle login of different user types. The parent has a reference to all their children, and the child has reference to all their parents. The reminder attribute of the Child class represents a datetime for when the user should be reminded to take action after a bad measurement, etc. The rest of the associated attributes connected to a child and a parent are handled with the EHRscape database, which implements the openEHR specification. These attributes include, but are not limited to: blood sugar level, birth date, gender, body weight, height. Thanks to the openEHR specification, this could be further extended to cover any health specific representation for the users.

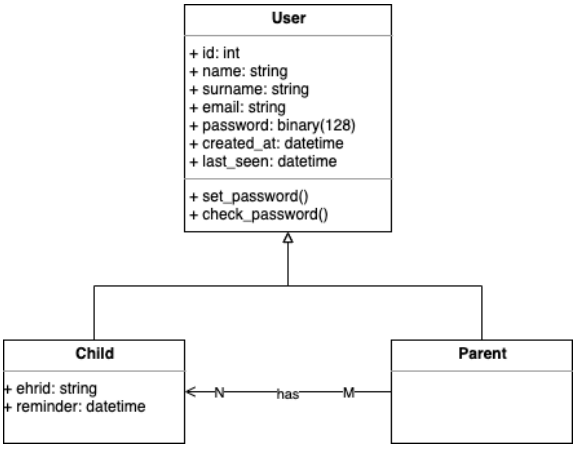

Figure 2 - Central Classes

Usage Patterns in React/Redux

The project, as previously mentioned, uses React in combination with Redux to handle state. The state is built on three cornerstones: auth, common and EHR. Each one is responsible for storing a different kind of data in the state. The auth stores temporary data, such as login input, password input, etc. These are cleared after a successful HTTP API-request has been made. The common part stores data regarding the current user with the children/parents, authentication tokens, etc. This part is updated after login, after requests which update the databases, etc. The EHR part stores data regarding specific EHR-data, such as family information, blood sugar levels, etc.

The Redux state is handled with a store. The state is immutable and can only be interacted with by dispatching actions to the store. The store contains reducers which reduce the actions dispatched, these reduces describe how the store should update the state. The actions can also dispatch asynchronous API-calls through the middleware, which will update the state through the store upon a successful response.

User Experience Design

The application has two different views depending on which type of user is currently logged in. The two options are either that the user is the patient (child) using the application to enter his or her blood sugar values, or that the user is the parent of the patient. In addition, there is a login view which is the same for all users.

Starting page and login view – When a user enters the application, he or she will first see the starting page, which displays some basic information about Healthify and the service. The user will also be given two alternatives, either to log in to their existing account, or to register and create a new account. If pressing the button “Registrera dig” to register a new account, the user needs to enter information about him- or herself, like name, e-mail address, and password. The registration option is only supposed to be for parents to use, because the parents will be able to add new patients to the system when they are logged in. If they instead press the button “Logga in” to login, the user is

then prompted to enter his or her e-mail address and password. After entering the correct e-mail address and password and pressing “Logga in”, the user is successfully logged in to his or her personal account.

The patient view – When login in as a patient (child), the user will see a view asking them to enter their values related to their disease. The other views the patient can see after logging in is statistics about their values, and a view which displays information about the caregivers and who has access to the patient’s data. One of the main views is the implementation of a digital twin, where the patient can see a simulation of theirself digitally based on the values entered to the system.

The parent view – When login in as a parent, the user will see a view about which child(ren) is/are registered under them. The parent user can press the button “Gå till översikt”, which will then display the child’s values and information about the child’s caregivers. The first page displayed when logged in as a parent also gives the opportunity to add more patients/children accounts to keep track of. When pressing “Lägg till barn” to add another patient, the user will have to enter information about the patient such as name, e-mail address, password, birthdate, gender, and disease.

8. Layers and Architectural Framework

Figure 3 - Systems Overview (below) describes the overall interaction between the different components in the project. The API-calls between the Flask backend server, web application, and the EHRScape server is done through the RESTful architectural style, only supporting simple CRUD-actions. The frontend and backend server are hosted on Kubernetes together with the PostgreSQL database server. The React frontend server is responsible for serving the minified HTML/JS/CSS artifacts over HTTP(S). The web application that is served from the React server is then run in the user’s web browser. The web browser then communicates with the Flask RESTful server and the EHRscape RESTful API server. In conclusion, this can be described as a three-layered architecture.

Due to time restrictions and restrictions from the customer, authentication is handled in a non-production ready way. Authentication to the Flask server is done through a JWT-token received after a successful login. This would in a production scenario be replaced with either login through Bank-ID or through the customer authentication system. The authentication to the EHRScape is done through the basic access authentication method. The username and password are provided to the user by the customer and is inputted by the user during the first use of the website on a new browser and are then stored in the Local Storage of the browser. This is not production-ready and would be implemented differently if the project was to go into a production environment. The authentication between the Flask backend server and EHRscape is similarly done through the basic access authentication method, but is in this case stored in Kubernetes secret storage. Everything within the Healthify box in Figure 3 - Systems Overview (below) is owned, developed and maintained by the project group, while the systems outside this box are merely used by the project group.

The PostgreSQL database server, the Flask backend server and the React frontend server are all served from Kubernetes. The PostgreSQL database requires persistent storage from Kubernetes, which the frontend server depends on. The frontend and backend servers are by themselves stateless and only depend on the PostgreSQL server and EHRScape server.

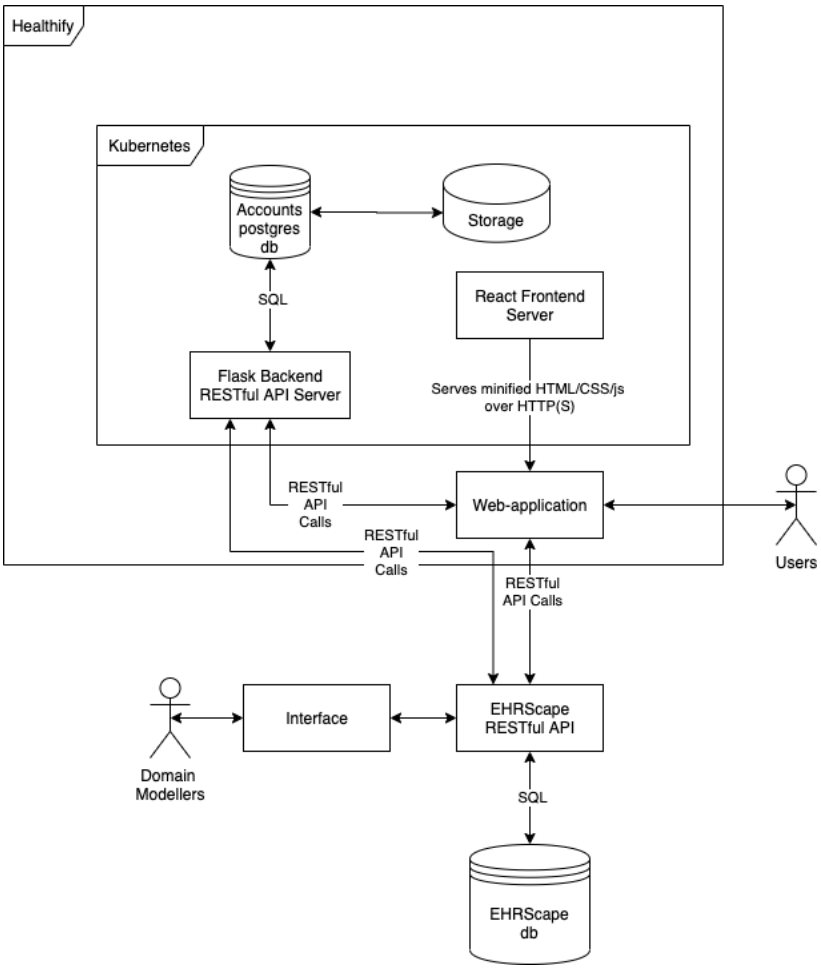

Figure 3 - Systems Overview

9. Changelog

Version	Date	Author	Changes
1.0	2020-10-10	David Forslöf Jesper Carlsson	First draft
1.1.1	2020-11-09	David Forslöf	Clarified Figure 2 – Systems Overview. Described Chapter 8 - Layers and Architectural framework more in depth. Described usage patterns in react/redux and central datatypes in Chapter 7 – Key Abstractions.
1.1.2	2020-11-11	David Nyberg	Added details in section 6, usability and efficient development. Also added section 5.1 regarding alternative design choices.
1.1.3	2020-11-14	Jesper Carlsson	Added User Experience design in section 7. Added architecturally significant requirements based on SRS
1.1.4	2020-11-19	Cecilia Persson	Proofreading for correctness and consistency + some changes to increase legibility/clarity.
1.1.5	2020-11-20/21	Jesper Carlsson	Reviewing document and changing in regards to comments
1.2	2020-11-21	David Forslöf	Cleared up 7. Key abstractions – User experience design
1.2.1	2020-11-22	Jesper Carlsson	Adding link to SRS and updating section 4
1.2.2	2020-11-23	Jesper Carlsson	Removing major parts and adding link instead to relevant document
1.2.3	2020-11-27	Jesper Jissbacke	Changed front page to match other documents.
1.2.4	2020-12-07	Cecilia Persson	Proofreading, changing to the passive voice.