Future Development

Klasson M

December 11, 2020

HeartByte

Current Version 1.0 [08-12-20]

Previous Versions

Version Number	Published Date	Description of revision	Author	Approved by
0.0	2020-11-29	Creation of document	Max Klasson	N/A
1.0	2020-12-08	Final version of document	Max Klasson	Patrik Palmgren

Contents

1 Purpose **4**
1.1 Description 4

2 Ideas for Future Development **5**
2.1 Saving a custom view to a database 5
2.2 Develop a dark mode function 5
2.3 Trend analysis 5
2.4 A more advanced notification log 5
2.5 Tweaking Rule Engine based on data set size 6
2.6 Rule-based Access Control and automation of assignment of permissions . . 6
2.7 Integration with other databases 6
2.8 Migrating from SQlite to PostgreSQ 7

3 Design Decisions **8**
3.1 Design Decision front end 8
3.2 Design Decision back end 8

4 Scalability **9**
4.1 Prioritizing of large data sets 9
4.2 Filtration of large data set 9
4.3 Increasing the number of users 9

5 Support and Maintenance **10**
5.1 IT-Governance 10

1 Purpose

This document aims to aid Region Östergötland in their future development and management of the product.

1.1 Description

As developers of this e-service, we are more than happy to assist Region Östergötland in future development. Our number one priority is the patients and that healthcare professionals can work in a safe environment. We want to show you that our product is easy to integrate and very expandable, both in handling the many but also in handling the few.

2 Ideas for Future Development

In this section, we will discuss ideas and suggestions for Region Östergötland in further development of our product.

2.1 Saving a custom view to a database

At the moment it is not possible to store the view created by the user anywhere. Storing them in a database for future use would be a great idea that would not be hard to implement.

2.2 Develop a dark mode function

The healthcare is a 24/7 service, which means that personnel works all around the clock. In order to support personnel during the night-shift, a dark mode functionality would a great idea.

2.3 Trend analysis

Due to the time constraint, the Rule Engine only classifies patients using the most recent measured value since this is simple to implement. This is however not the best solution, since the customer has mentioned that absolute values are often not the most relevant factor when prioritizing patients. The customer thinks that individual patient trends are more relevant. Originally, this was in the scope of the project but there was not enough time to implement this.

2.4 A more advanced notification log

Twp simple notification logs are implemented in the system. The first contains information about all patients handled by the user and the second is a log for a specific patient. The second contains a full log of all events, but the overview notification log for all patients is simplified. Thus the simplified log could be improved and this is discussed below. In the system now, after the overview data is retrieved to the overview module, this data is then passed to the notification log. The log displays date of the latest measurement added for a patient, as well as which measurements were added this date. It is only the last measurement for a patient that is displayed since it would be slow to send all data for every patient to the front end, and then have the front end derive notifications from this. A better solution would be an implementation closer to the Trend Analysis module

2.5 Tweaking Rule Engine based on data set size

As the number of patients increase, there is a high likelihood that the Rule Engine will prioritize a large number of patients as the highest priority. In this case it will be quite hard for a health care 24 worker to prioritize between the patients. To handle this there are two possible approaches. Dynamic prioritization The absolute number of patients classified as the highest priority class is limited. Thus, only the e.g. 20 most critical patients are classified as the highest priority. This would help the system avoid notification fatigue in its users. Non-integer prioritization In the system now, the classifications are made in integers. One integer corresponds to one group of patients with a condition classified as equally serious. Non-integer prioritization would change this, and instead classify a patient's condition into a non-integer number greater than zero using some mathematical formula. There are some problems with both approaches. The first creates a invalid view of the patients. If there are many critical patients, this would not be reflected in the overview as some would be classified as less severe. The second approach makes prioritization less clear as a decimal number is harder to interpret than a class, such as "Critical condition" or "Low priority". The best solution would be a combination of the two. In this, the classification into priorities is dynamic so there is a fixed maximum number of patients in the most critical classes. However, next to the symbol representing the class the non-integer number is also displayed. Thus the user can both get a good overview with groups with a graphical symbol or color as well as a number so the user can differentiate between the members of each class. Implementing this would require a re-work of the Rule Engine as well as the database storing the custom rules for the Rule Engine.

2.6 Rule-based Access Control and automation of assignment of permissions

In the system now, the permission scopes of users are assigned manually in Auth0's dashboard. This solution is not scaleable, since it will require a large amount of work to add all the users and their permission scopes. A better solution would be to implement RBAC: creating roles with a set of permission scopes and then assigning the roles to a user. In RBAC you analyze the needs of users and group them into roles. Since patient access is based on what HCU you work at and some functionality is limited to admins, it is simple to split users into groups and to see what access roles these groups require. To further automate this solution, the roles should be automatically assigned to a user after looking them up in the personnel database.

2.7 Integration with other databases

In discussions with the customer, two other databases were mentioned: both the Swedish Population Register and the customer's personnel database. In the future demographic patient data would be retrieved from the population register, which would ensure that the data in correct. The customer's personnel database would replace the personnel database in the HeartByte back end and the customer's database would be used for the RBAC illustrated in Figure 12. Since HeartByte does not have any knowledge regarding the complexity of

retrieving data from the Swedish Population Register, it is hard to estimate how much effort would be required to replace the demographic data source. HeartByte does not have any information regarding the structure of the customer's personnel database, it is also difficult to estimate the effort needed to replace the current database. Since this database most likely is not very complex, this should be relatively easy

2.8 Migrating from SQlite to PostgreSQ

Due to the simplicity of setting up and managing the database, HeartByte opted to use SQlite to be its database system during the development process. SQlite is commonly used for development since it does not require a separate server to host the database and is easy to use. SQlite has some weaknesses that are not ideal for full scale deployment however: it has poor performance on larger data sets, no real access control and does not enforce type checking. PostgreSQL is more complex to set up and requires a separate server, but performs much better with larger data sets and more complex queries as well as being more strict in enforcing constraints. It also has features to support access control. This is why it is usually preferred for deployment. Migrating from SQlite to PostgreSQL could pose some issues since PostgreSQL applies stricter rules than SQlite. The fact that SQlite does not enforce type checking and that implementing SQlite with SQLAlchemy results in that foreign key contraints are not enforced could pose a serious issue when migrating. To mitigate this risk foreign key constraint enforcing was manually added to the SQlite database using the command below. This should make the migration much easier, if it is wanted in the future.
PRAGMA f o r e i g n k e y s = ON;

3 Design Decisions

In this section, our design ideas will be presented. We hope this will provide you with our way of thinking.

3.1 Design Decision front end

The front end will be developed using the JavaScript framework ReactJS. This is due to a few reasons:
(i) By building a web app and using React, platform and OS independence is enabled.
(ii) Due to the skill constraint, we want to leverage existing knowledge. The team is proficient in JavaScript and React is a JavaScript framework.
(iii) Due to the time constraint, we need a language that is quick to learn and simple to use. This holds true for React, which due to its wide spread use has many tutorials and good documentation.
(iv) One of the architectural goals is to achieve high modularity through reusable components.
React facilities this through allowing developers to split the UI into independent components and there are many free open-source components because of the popularity of the framework enabling quick development.

3.2 Design Decision back end

Due to the time and skill constraints, this back end and the databases would ideally leverage existing knowledge in the team. The back end will be developed in Python, using the web framework Flask for the server and the framework SQL Alchemy for the databases.

4 Scalability

In this section, the scalability of the product is discussed.

4.1 Prioritizing of large data sets

The Rule Engine is placed in the Front end logic layer, as seen in Figure 3, due to the reasons outlined in section 3.6. The Rule Engine only prioritizes patients based on the most recent measurements, so the classification is not computationally heavy. Due to this, a large number of patients should not affect the performance of the system in a major way. The performance would although be worse if the Rule Engine would implement more computationally heavy classification rules on a large data set, as mentioned in section 6.4.1. However, with a very large data set and many patients displayed in the Patient Overview another potential problem would be created. If there are e.g. 100 patients displayed unfiltered, there might be many patients classified as high priority. Then it would not be easy to see which ones are the most relevant. This can be addressed by the user, in the existing system, by filtrating on some factors to reduce the number of displayed patients.

4.2 Filtration of large data set

Since the filtration functionality is placed in the front end, performance could be expected to suffer when working with very large data sets. The rationale laid out in section 3.6 is based on assumptions of a moderately sized data set and frequent filtration. However, if the data set would be very large and the filtration isn't happening frequently then the rationale might not be valid. The time lost due to calls to the back end and EhrScape could be offset by time lost due to hardware limitations on the client side. It is considered unlikely that this will be the case however, since simple tests on larger sets of patients have not shown a drop in performance. If however performance were to be affected due to a larger number of patients, the recommended solution would be to limit the number of patients that are returned to the overview. In the current implementation, all patients that the user has authority to access are returned. This could be changed so that only patients tended to by the user's department or team are returned. To implement this, additional data fields in the demographic database in EhrScape could be added.

4.3 Increasing the number of users

Increasing the number of users should not affect the performance of the system, since the system is designed utilizing a two-tier fat-client where no computationally heavy functionality is implemented in the back end. However the process of manually adding and updating permission scopes in Auth0 to each user when they change or register for the first time will be time consuming for an admin.

5 Support and Maintenance

In this section, the maintenance and relevant governance frameworks is discussed.

5.1 IT-Governance

The E-service supported by PM3
Our product works well if you are using PM3 as your it-governance framework. Managing the product together with your previous organisational components is a strong feature and relevant for optimal portfolio management. Furthermore, Pm3 will allow our product to support the organisational processes already in place, preventing the product from becoming an isolated process.

The E-service supported by IT-IL
It is essential to view our product as a part of the whole organisation, therefore we advice to perform continual service improvement. Our product, which in itself aggregates data, makes business decisions easier because you have this quantifiable data at hand.

Superuser
We recommend that you educate superusers for our product, they will assist in the integration of the product. Preferably this will be done before delivery.

Introduction
Our product is a stand alone application so we recommend that you phase out the old system or use our product as a support system.

Service level agreements

- Continuous back-up during the implementation phase.
- In case of a crucial event the product will be functional within 24 hours.
- Personal data will be handled according to GDPR or PDL.
- The number of patients the application can handle is 20.000.