ETiCCS application

built in React-Redux

Team: No Idea

- Adam Alsegård, A0153097B
- Carlos Garcia, A0164527B
- Timoté Vaucher, A0153040Y
- Sheng Xuan, A0142230B

For instructions on how to set up the project, please see 'README.md'

Overview of system

The system is divided into frontend (client) and backend (server). The frontend starts at client/index.js where the Store, Provider, Router and App are set up and connected. The root component is App.js, which handles the routing of all parent UI components, while store.js synchronizes the store with the backend and database.

Every UI is then built with presentational components that only contains static elements. The components get their functionality via containers. The containers connect the dumb components to their props and dispatch functions. When a function is dispatched it triggers an action which in turn triggers the root reducer. The root reducer routes the action to the corresponding reducer which in turn updates the store of the affiliated data structure. The store syncs with database and then sends the updated state to all subscribing components. And by that the circle is complete.

The backend starts at server/server.js where it sets up the Express server and create a connection to the RethinkDB. On one hand, for non-API request, it serves the same base file containing an entry point for the front end. On the other hand, when it receives API requests, it routes them using the API router to different controller that will themselves call the model in order to update the database.

Folder structure

- **package.json**

Holds project information and defines all the dependencies.

- **dbSetup.js**

Connect for the first time to the database, creates it and add the necessary tables.

- **webpack.config.js**

Defines the rules for our middleware and hot loader.

- **client/**

Contains all files that works on our frontend.

  - **actions/**

    Files with all the actions for the different UIs. One file for each UI. Also contains actions for the Login/Register logic.

  - **components/**

    Dumb presentational components, no functionality. One component for every UI.

    - **forms/**

      Contains the data for the forms. If a UI has multiple forms they are collected in a specific folder.

    - **helpComponents/**

      Base components to be used in multiple containers.

    - **Lab/**

      Contains the dumb components for the three lab UIs.

    - **Pathologist/**

      Contains the dumb components for the two pathologist UIs.

  - **containers/**

    Maps logic to the dumb components, connects them to props and dispatch functions. One major container for every UI that maps the data to the displayed table.

    - **forms/**

      Maps correct state and dispatch function to the form components. One container for every form.

    - **communityUI/**

      Contains the containers for the table components in CommunityUI. Connects the dumb helpComponents and/or forms to its correct functionality.

    - **gynecologistUI**

      Same as above but for GynecologistUI.

    - **householdUI**

      Same as above but for HouseholdUI.

    - **labUI**

      Same as above but for LabUI.

- **nurseUI**

  Same as above but for NurseUI.

- **pathoUI**

  Same as above but for PathologistUI.

➢ **reducers/**

Updates the store/DB according to the dispatched actions. One reducer for every type in the store (community, household, patient, sample). Also includes reducers for the Login/Register functionality.

- **AppReducer.js**

  Root reducer. Combines all the reducers and chooses the correct reducer for a specific action type.

➢ **routers/**

High-level component that describes the basic layout of a UI.

➢ **style/**

Contains the default CSS for the application.

➢ **util/**

Defines helper functions and api calls to be used in actions and containers.

➢ **App.js**

Parent router. Contains all the other routers and therefore their components as well. Directs the most high-level connections.

➢ **index.js**

Root of application. Creates the initial store and sets up the Redux application.

➢ **store.js**

Configures the store and connects it to the database and middlewares.

❖ **server/**

Contains all files that works on our backend.

➢ **api/**

Handles database calls, model part of the application.

➢ **controllers/**

The logic for the api calls that handles http requests.

➢ **routes/**

Routes http requests to their controllers.

- **util/**

Utility functions for cryptography and mailer (used to retrieve lost passwords).

- **index.html**

Our base HTML root. Defines what stylesheets etc to load.

- **server.js**

Creates the server, handle the connection with the database, defines the rules of serving the content to the user.

❖ **docs/**

Contains the generated documentation of all code.

- **config/jsdoc.json**

Defines how the documentation should be generated.

System planning

The first challenge of the project was to define the component hierarchy for the different user interfaces. To do this the requirements had to be specified. This was a bigger challenge than expected because the given requirements, data modules and project description sometimes contradicted each other. However, all UIs had in common that they showed the data in tables. And the tables in turn were composed of only a few different components. The component hierarchy was therefore based on this table-focused approach. An example of the hierarchy mock-ups can be seen in Appendix 1.

The library ‘sematable’ was used to implement the tables. This helped with a lot of the search and sorting logic but also brought on other difficulties in how to define and connect components, especially when they only should modify one row. Because the tables often were composed of the same type of components a number of ‘base components’ were implemented to keep a modular approach. Among these components were templates for Checkbox, Select input, Text field, Link, Delete button, Edit button and Edit form.

To connect the components to their functionality containers were used. One container is only responsible for one specific component or functionality. This led to a large amount of containers in the project. For example every form needed two containers. One to map the props and dispatch functions to the form component itself and one to wrap the first container and connect it to the correct button.

State tree

The state tree consist of 6 different levels; ‘community’, ‘household’, ‘patient’, ‘sampleHPV’, ‘sampleBiomarker’ and ‘sampleBiopsy’. Community is the root, here we store the general information about all households in a specific community. Every entry here is therefore a household. When the volunteer clicks on a household the corresponding family is loaded.

Every entry in the household data structure is a person. The information stored here corresponds to the information from the forms in Household UI. If a person is found eligible for study then the volunteer register that person as a new patient with the ‘Registration’ form. Some information (as name, age, patientID etc) is copied from the household to the patient data structure. After registration there are a few more forms for volunteer to fill out, where the last one is the one where user collects a HPV sample and creates a new entry in sampleHPV.

The volunteer can only edit a certain form in Household UI if certain requirements are met. For example will the ‘Age-Eligible form’ only be enabled if the person is female and 19 years or older. ‘Eligibility form’ is enabled if the woman is present and willing to participate. ‘Registration form’ will only open up if participant pass all the medical requirements and the woman is deemed eligible for study and so on. The patient ID for the created patient will be the same as the person’s National ID and sampleHPV’s sample ID will be the same as the barcode for the submitted sample. The ‘Questionnaire #1’ checkbox will be ticked after all forms are completed and ‘HPV sample’ checkbox will be ticked if a HPV sample was collected that day. All data from the forms are stored in the database, even if it’s only a small fraction that’s actually displayed in the tables. The same logic for forms are implemented in all the other UIs as well.

The samples were separated into different data structures because they do not follow the same flow and therefore don’t need the exact same fields. HPV sample is created by volunteer, tested by lab technician and read by Nurse whereas Biomarker and Biopsy samples are created by nurse and gynecologist respectively, prepared and stained by lab technician, interpreted by pathologist and read by gynecologist again. The samples belongs to a specific patient and therefore inherits patient ID and name.

The patient data structure is used and modified by nurse, gynecologist, lab technicians and pathologists. It binds together most of the data flow after that a person is deemed eligible. Each patient object contains all the information collected by different forms, many of the attributes are not displayed in the table, but users are able to see the stored attributes when they view the corresponding forms because the form loads its initial values from the database and display them in the fields.

Data Flow

The data flow of this app is defined by the requirements given. Basically, we need to handle the flow of patient data and test sample data.

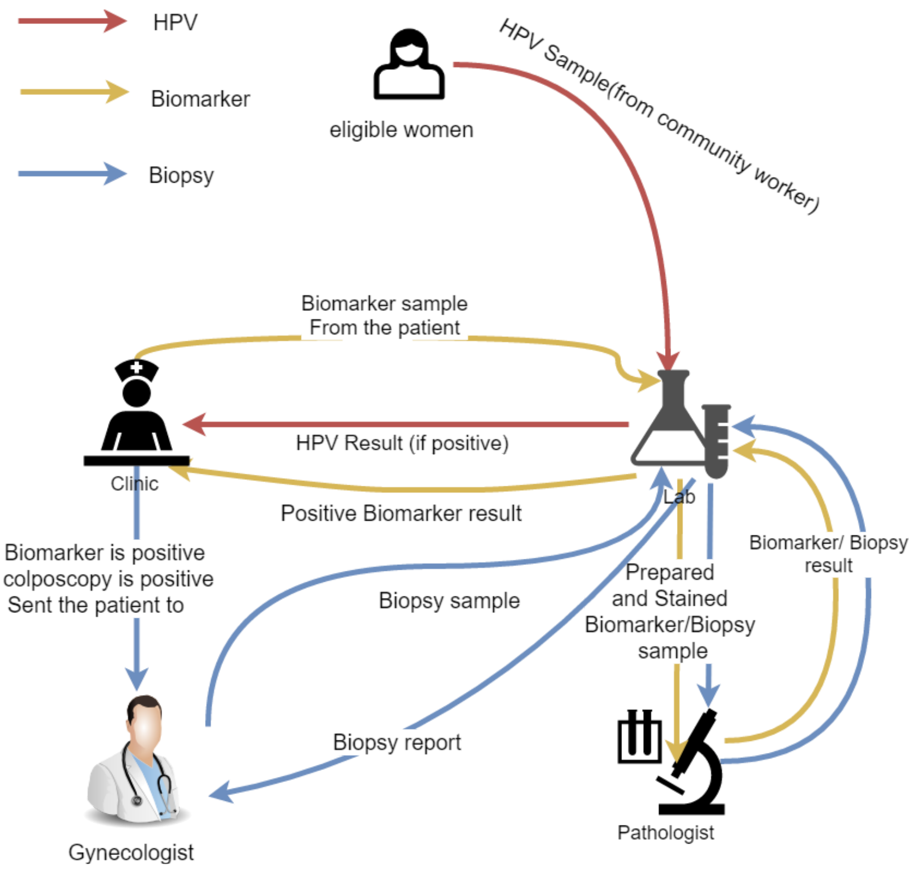
HPV
Biomarker
Biopsy
eligible women
HPV Sample(from community worker)
Biomarker sample From the patient
HPV Result (if positive)
Clinic
Lab
Positive Biomarker result
If Biomarker is positive or colposcopy is positive Sent the patient to
Biopsy sample
Prepared and Stained Biomarker/Biopsy sample
Biomarker/ Biopsy result
Biopsy report
Gynecologist
Pathologist

Fig.1 Data Flow

Fig.1 demonstrates how the patient and test sample data flow. Each person in a household needs to complete some forms to check whether she is eligible for the HPV examination. Community workers directly send an eligible female's HPV sample to the HPV lab. Then the HPV lab will add the test result of this sample, if HPV sample is positive, this female will be added as a patient to the patient list, which can be viewed in the nurse UI. The nurse can then give the patient a colposcopy examination and take a biomarker sample from that patient. The biomarker sample is sent to the biomarker lab, after this sample is prepared and stained by the biomarker lab, the test sample is sent to the pathologist. Pathologist can edit the test result of a biomarker sample. If a patient has either positive result in the colposcopy examination or positive biomarker result, patient's information will be sent to gynecologist for the final examination and treatment. Gynecologist can take biopsy test sample from the patient and send it to the biopsy lab, similar as biomarker sample, this biopsy sample is prepared and stained by the lab and interpret by the pathologist, the final result is sent back to the gynecologist.

Division of work

All members started by discussing the component hierarchies and the requirements. A common understanding was achieved before everybody took on some UIs to define the component hierarchy for. After that a boilerplate was implemented so everybody had the same foundation to start from.

Because different team members would leave the country in different rounds no consistent responsibilities were set up from the beginning but instead new tasks were distributed every week for the available members.

After the boilerplate was set up and requirements clearly defined the basic UIs were implemented by the available members. This included a template for how to implement a sematable, base components and an example of the data flow with containers, actions, reducers and routers.

The next step was to implement the forms. A couple of experiments with multi-step forms were carried through before it was found that the forms needed to be separated to handle the functionality correctly. Guidelines for how to document the code was set up and automatically generation of documentation was enabled with JSDoc.

Parallel to all this was the development of the backend, which one member worked on. The database was set up with RethinkDB and the backend make use of Express.js for its logic.

In the end all different parts were combined, the data flow between all UIs ensured and tests were implemented for add, edit and delete person in Household UI.

Github was used for version control and all members worked on different branches. When a feature was finished a pull request was submitted and examined by other team members before being accepted and merged to master. This ensured a fast and secure workflow with few merge conflicts.

UI Design

The use of sematable and react-redux-form greatly facilitated the styling of the application. They provided an easy and clean way to implement and run searches through the system and each individual table. They were helpful in sorting the tables as well as to implement the forms in an efficient manner. These forms were also chosen to appear as though they popped up from the top of the screen with a fade and fade out effect to give the system a cleaner look and be more user friendly.

Confirm alerts were also put in place to make sure that the users did not accidently delete patients. The forms and tables were all given a similar look in order to keep the system consistent and user friendly. These forms also included required fields and validation so that the inputs would not be unrealistic (eg. dates set in the future) or that the users would not accidently skip a field.

A few styling customizations were implemented at the very end using Bootstrap and css for a more visually pleasing result.

UI testing

Three unit tests were implemented for the add, remove and edit features in the household interface. These tests were chosen as they are very basic features which should be implemented in the system to be able to add and update the status of the people. The test for add person first creates a newPerson object which contains the information that would be passed when a new person is added to the household UI. This is required by addPerson in HouseholdActions along with 2 ids that are manually set in the test. This person data is taken by addPerson and the value for eligibleAsking is calculated by determining if the person is a female over the age of 19. It compares this to the eligibleAsking set to true and the rest of the information that was set in the test.

The test for editing a person is similar to the one for adding a person. It requires an object which represents what the new fields will be set to and also calculates if these new values change the status of eligableAsking. The test compares these values with the ones that are returned by editPerson and checks that the eligibleAsking is now set to false as the value of the gender was switched to male in the data. The test for deleting a person only requires a single argument to determine which patient will be removed.

These tests are run using jest and requires that “$('#' + formID).modal('hide');” be removed from HouseholdActions as the tests cannot run properly when this is present. This is needed for hiding the forms in the UI, however the actions are still completed.

Team Contribution

Adam Alsegård:

- Define component hierarchy for Community, Household & Pathologist (x2)
- Implemented interfaces and logic for Community, Household, Nurse, Gynecologist.
- Set up guidelines for how to generate documentation.
- Implemented forms for Community and Household. Finished Nurse forms.
- Implemented data flow between household and patient

Carlos Garcia:

- Started of Nurse forms
- Tests

Timoté Vaucher:

- Define component hierarchy for Nurse & Gynecologist.
- Boilerplate to get started everybody started.
- Implemented model forms to use as template.
- Backend handling.
- Login/Register/Lost password pages.

Sheng Xuan:

- Define component hierarchy for Lab technicians (x3).
- Implemented interfaces and logic for Lab technicians (x3) and Pathologist (x2).
- Implemented forms for Lab, Pathologist and Gynecologist UIs.
- Implemented data flow between patient and test samples, filter correct data to be displayed in different UIs.

Appendix 1:

Mock-Up of a Table UI:

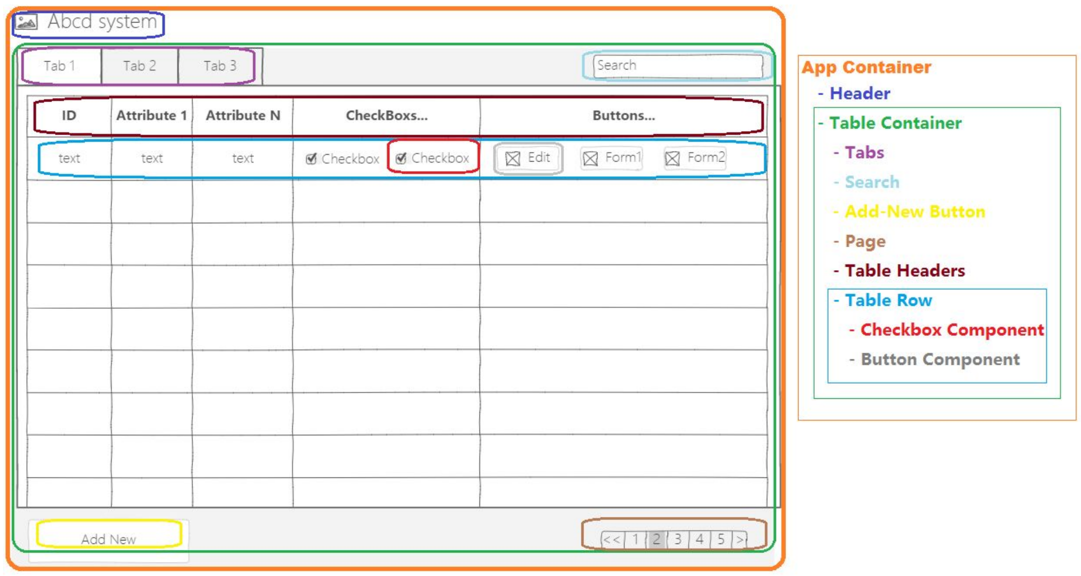

Fig.1 UI Hierarchy

Current UI:

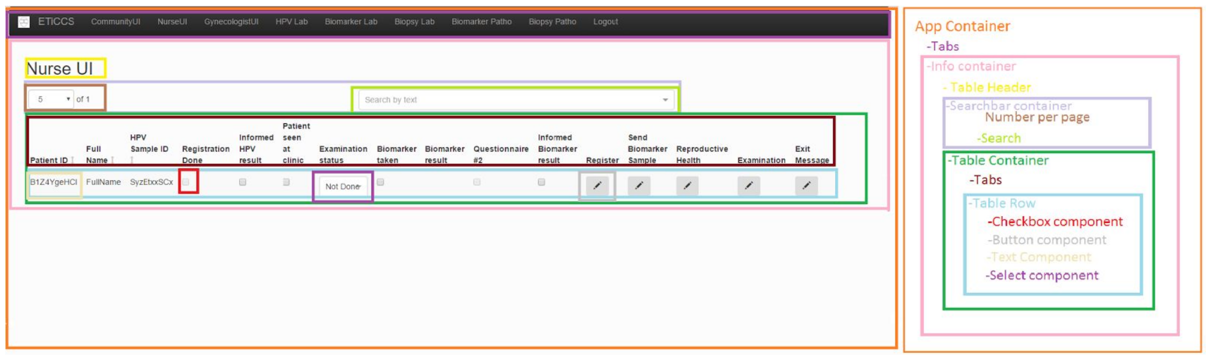

Fig 2. UI Hierarchy II

Our initial UI hierarchy included a large table with various different fields and tabs to navigate the different UIs (Fig 1). Later, we chose to use 'sematable' for all of the UIs to give the system a sleeker look (Fig 2). Therefore, since each UI uses the same styling, the UI hierarchy for each is almost the same

There are two parts in the App container: the tabs of each UI and the data table. The tabs enable the user to switch between different UIs, for example, a lab technician needs to switch among HPV samples, Biomarker samples and Biopsy samples. Within the table, there should be SearchBar and Page components to help the user check the data they want. For some views, there is an 'Add New' button to enable the user to add new rows into the table. Table headers (titles) are defined as props for each table. For each table row, there will be both props and states. Props items are generally those static and not editable columns, and components including editable checkboxes and buttons in the row will be states, because the user needs to manipulate these components to edit the data in the database. Basically, checkboxes are used for toggle some status of the data, such as sample is received. On the other hand, buttons are used to open a particular form for the user to enter some data, while text fields indicate the data values being being stored.