SW Engineering CSC648/848

Project Remedium

Team 4

Spencer Heltsley - sheltsle@mail.sfsu.edu
Kevin Chan
Zi Collin Zhen
Kolapo Agunbiade
Andrew Keelin

Milestone 4

December 7th, 2020

History Table:

Submitted: December 7th, 2020
Revised M4 Submitted: December 17th, 2020

1. Product Summary

Our project has stuck with it's codename through development, and we've decided to name it that, as it seems to have stuck fairly unanimously: Remedium, which stands for medicine, or healing, in Latin. On release, our product's committed functions should be:

- Profile creation, displaying the profile of a given PT and/or Patient to allow for information to be easily displayed and accessed
- Comprehensive patient listing, allowing for patients to easily be searched, discovered, and their routines interacted with/assigned/reviewed
- Listing of video content within a content library, allowing PT's and patients an easy way of quickly linking their own video of their procedures, for comprehensive review by the relevant parties
- Exercise routine creation, allowing for customized series of exercises to be created and easily linked to individual patients, and their reports on such routines easily reviewed
- Comprehensive care tracking, keeping careful track of how much time a PT has spent interacting with a given patient, and doing what, so that the Patient has a good idea of the quality of their care
- Reporting of all previous data upon request, in the profile page of a given patient for easy review by all relevant parties
- Text-based communication between patient and PT via a real-time connection between logged accounts

We are working on some other things, such as line tracing over videos, and a notification queue that should be somewhat unique, but as is, those should be our only truly unique features. All other features are designed to be simple to use, easy to learn, responsive, and a generally smooth user experience. Our website features an incredibly easy to use UI that allows for PT's to navigate quickly and seamlessly to any information that they might need, and keeps all that information relevant to the person originally navigated to, so that no messy information might be mis-assigned. Our system works in a way that ensures everything should be apparent at a glance, at all levels of depth, to streamline the user experience and ensure that a quick, responsive, and painless transition is possible for users of all levels of technical skill. Our website can be accessed via the following link:

http://remedium.fit/

2. QA Test Plan

Unit Test Plan:

Our unit test plan is primarily aimed at the functions of reporting information, and assigning the exercise routines. Due to our hardware setups, we have a fairly comprehensive testing plan for every single thing that we upload. Our software is tested on Edge, Chrome, and Safari at varying stages, and any issues are fixed during the initial stages, depending on the error. Browser-based errors are fairly simple to fix, as they tend to generate their issues in the console of the browser itself, when the elements are inspected. Our hardware setup is also fairly unanimous. Whichever branch we're pulling from will be discussed and updated, and then each pull undergoes the same steps: Visual Studio Code is opened, and the packages are navigated to. The proper npm installs are performed as per the package.json file, and the versions are checked to ensure they are in alignment beforehand (we've had 4 hour issues before solved by simply reinstalling dependencies). The proper backend servers are started, depending on the props being tested, and then, depending on the browser, the proper plugins are enabled. For chrome, in particular, we get strange errors when querying other localhost addresses, which tends to ruin functionality. For this particular issue, we have an anti-CORS protocol plugin that we enable when performing tests on that function. It is assumed that none of our individual components that do not interact with each other will conflict.

Unit testing originally was going to use tools, however, it has not done so. Our individual components are generally rendering things on their return, and after many hours of testing, with several different tools, we were unable to devise a way to get the test functionalities to pick these up. However, our testing plan is saved by the fact that we are component based. This allows us to test each individual return in each file as it renders, and verify that it is functional (without outside interference from the backend) by querying from sample json files to get their information. This is the way we have tested everything, procedurally, before uploading it. In addition, most of these operations log to the console.log within the IDE, and therefore, any issues within the data show up there, and allow us to verify everything going in and out.

Integration Test Plan:

Utilizes the exact same setup with hardware and software as described above. User Stories 1 and 3 will be used for this testing. For the sake of convenience, they will be summed up below:
1: John, a 50 year old technical neophyte, but experienced physician, who's determined to get a better use out of his telecommunications technologies. He's very good at working in person, but inexperienced at remote treatment and communication.
3: James, a 30 year old, fresh out of school professional, who has used technology his entire life, but desperately craves a separation between his work and home life. Because of his experience, he'll be more demanding of the UI to be modern.

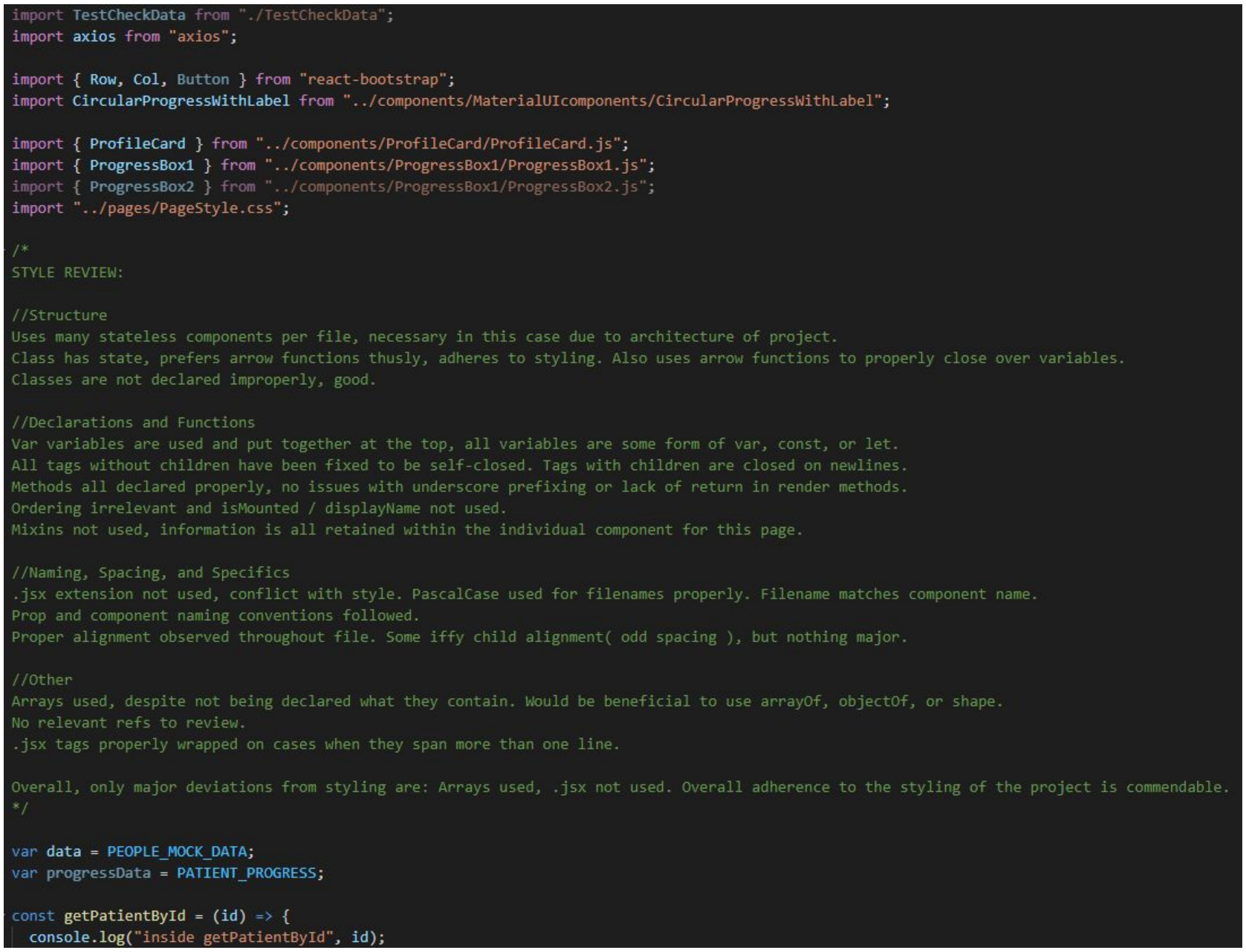
Test ID	Description	Date Tested	Steps	Prerequisites	Test Data (if needed)	Pass/Fail
1 Chrome	Navigate through each element within the navbar at the top, then into 2 discrete users to verify all information regarding them renders correctly.	12/7	Move to each tab and check to make sure each is quick, descriptive, and accurate.	npm install User should be logged in as PT, and must start from the home page.	No Data Passed / Input besides Default Profiles Near Instant Results	Pass
1 Edge	Navigate through each element within the navbar at the top, then into 2 discrete users to verify all information regarding them renders correctly.	12/7	“	npm install User should be logged in as PT, and must start from the home page	No Data Passed / Input besides Default Profiles	Pass
2 Chrome	Navigate to Patient Profile, and use the report tab to generate a report for each of 3 different patients, verifying that the information in the report is accurate to the patient information in the database.	12/7	Generate report for first 2 patients in list, review info, and verify completion of tasks is calculated correctly. Then move to a random patient in the list, and verify completion of tasks is calculated correctly.	npm install Start server for backend database querying User should be logged in as PT, and must start from the Patient Search page.	All Reports Generated Correctly All Elements Calculate correctly	Pass

2 Edge	Navigate to Patient Profile, and use the report tab to generate a report for each of 3 different patients, verifying that the information in the report is accurate to the patient information in the database.	12/7	“	“	“	Pass
3 Chrome	Test sending messages back and forth between several different chats, ensuring that crosstalk does not occur, and that the data of each message is accurately and promptly delivered and displayed.	12/7	Open chat in 2 tabs, navigate to 2 different rooms, and verify chat works in both. Then, using 2 more tabs, enter each of these rooms and send different test messages, making sure each room only gets its messages. Organize tabs to prevent errors.	npm install Start messaging server Disable CORS in browser if testing locally, will not send requests on chrome otherwise. Must start at the dummy room selection screen, or by inputting a specific room in the URL.	All Elements Render/Send Correctly in both rooms, regardless of message Length. CORS protocol present in Google Chrome prevents delivery of requests. Unsure if this is a local issue. Rooms used: “test1”, “test 2”	Conditional Pass

3 Edge	Test sending messages back and forth between several different chats, ensuring that crosstalk does not occur, and that the data of each message is accurately and promptly delivered and displayed.	12/7	“	npm install Start messaging server Must start at the dummy room selection screen, or by inputting a specific room in the URL	All Elements Render/Send Correctly in both rooms, regardless of message Length. No CORS errors when testing locally. Rooms used: “test1”, “test 2”	Pass

3. Code Review

The coding style that we chose was the airbnb coding style, reviewable here on the github. We chose this style, primarily, because it allowed each of us to code in a very natural way, as it was fairly similar to the way we already coded. Because of this, it has been relatively self-enforcing, as it came to each of us very naturally. Some of the ways it may be further enforced upon review, are the revision of files based on certain functions being included, or certain libraries. The review has been conducted on the **ReportPageReviewed.js** (I wrote .c in the previous version of this because I was writing C code at the time, apologies) file accompanying this pdf in the github folder for Milestone 4. The review that has been conducted, was conducted on the original file ReportPage.js, which renders all components for the portion of patient profiles concerning their reports on their progress thus far. As it is a rather large file, I have opted against mass screenshots (and the pull request has confused me). Therefore, I will include a screenshot below of the results of a thorough, line by line analysis of this file, according to the airbnb React styling guidelines on Github. Lines with specific issues or notable things have been tagged with a //REVIEW: at the end of their relevant code, however, there were not many of these, and very few errors were found, mostly in the usage of arrays, and the file type being .js instead of .jsx.



```
import TestCheckData from "./TestCheckData";
import axios from "axios";

import { Row, Col, Button } from "react-bootstrap";
import CircularProgressWithLabel from "../components/MaterialUIcomponents/CircularProgressWithLabel";

import { ProfileCard } from "../components/ProfileCard/ProfileCard.js";
import { ProgressBox1 } from "../components/ProgressBox1/ProgressBox1.js";
import { ProgressBox2 } from "../components/ProgressBox1/ProgressBox2.js";
import "../pages/PageStyle.css";

/*
STYLE REVIEW:

//Structure
Uses many stateless components per file, necessary in this case due to architecture of project.
Class has state, prefers arrow functions thusly, adheres to styling. Also uses arrow functions to properly close over variables.
Classes are not declared improperly, good.

//Declarations and Functions
Var variables are used and put together at the top, all variables are some form of var, const, or let.
All tags without children have been fixed to be self-closed. Tags with children are closed on newlines.
Methods all declared properly, no issues with underscore prefixing or lack of return in render methods.
Ordering irrelevant and isMounted / displayName not used.
Mixins not used, information is all retained within the individual component for this page.

//Naming, Spacing, and Specifics
.jsx extension not used, conflict with style. PascalCase used for filenames properly. Filename matches component name.
Prop and component naming conventions followed.
Proper alignment observed throughout file. Some iffy child alignment( odd spacing ), but nothing major.

//Other
Arrays used, despite not being declared what they contain. Would be beneficial to use arrayOf, objectOf, or shape.
No relevant refs to review.
.jsx tags properly wrapped on cases when they span more than one line.

Overall, only major deviations from styling are: Arrays used, .jsx not used. Overall adherence to the styling of the project is commendable.
*/

var data = PEOPLE_MOCK_DATA;
var progressData = PATIENT_PROGRESS;

const getPatientById = (id) => {
  console.log("inside getPatientById", id);
```

4. Adherence to Original Non-Functional Specs

Application shall be developed, tested and deployed using tools and servers reviewed by Class TA in M0 (some tools may be guided by TA in the class, some may be chosen by the student team but all tools and servers have to be reviewed by class TA).
DONE

Application shall be compatible and usable on PC browsers.
DONE

Data shall be stored in the team's chosen database technology on the team's deployment server. **DONE**

Application shall be easy to use and intuitive. **DONE**

The code base should be well maintained so that new engineers can easily read and continue building on the code. **ON TRACK**

The site loading time shall be less than 2 seconds for all screens. **DONE**

The total data storage allowed by the web site shall not exceed 80% of the server capacity for this site. **ON TRACK**

The web site shall be capable to handle at least 50 users. **ON TRACK**

The web site shall be prepared to support scalability for adding future features. **DONE**