Coast Capital Contractor Records Database System - Internal Design Doc

The Terminal

November 29, 2017

Contents

1 Introduction

1.1 Overview

The aim of our project is to simplify the system Coast Capital uses for capturing contractor data. Their current system requires tedious manual data entry and complex spreadsheet manipulation, and we want to provide a better method of visualization, management and maintenance. We will be building an online web application designed for ease of use and powerful data analysis. The application aims to make it a straightforward process to add, edit, and visualize contractor data.

1.2 Goals

Our application will be easy to learn for new users such that there is not a long ramp-up period for using it in comparison to using excel spreadsheets. Furthermore, since our application is an online web tool, all the contractor data within it will always be the most up to date data available so no two users are viewing different data. The goal will be to implement the following features: a data filtering system, reports visualization, the ability to add or edit contractor data, and an admin panel for admin-specific tasks.

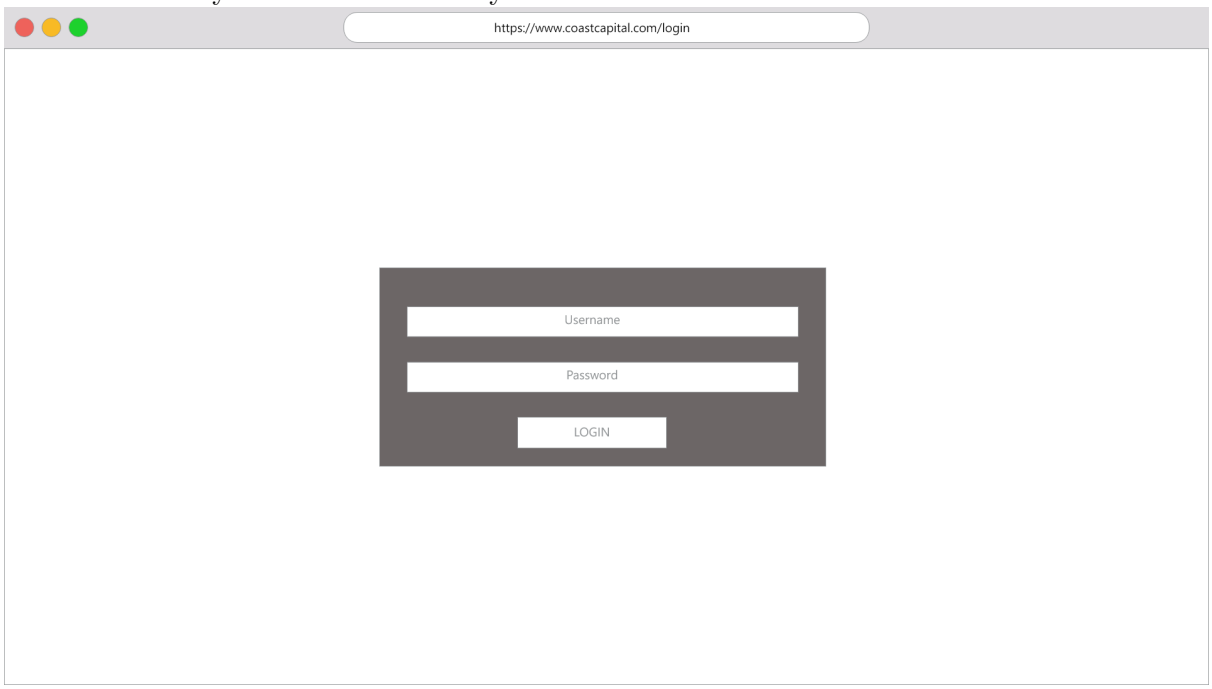
1.3 Assumptions

1. An admin does not require a filtering method for the various tables that they can access.
2. The FX table can be updated every 24 hours.
3. An admin can add as many skills as they want in the skills table.

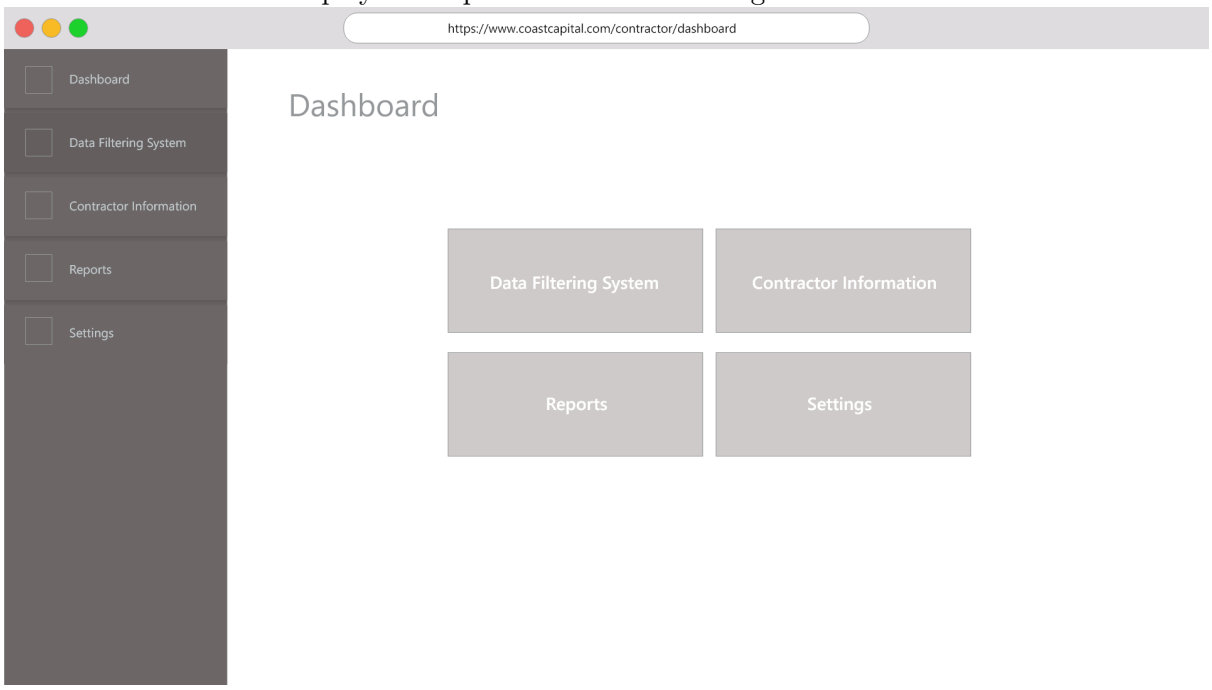
2 UI Design

2.1 Mock-Up Screens

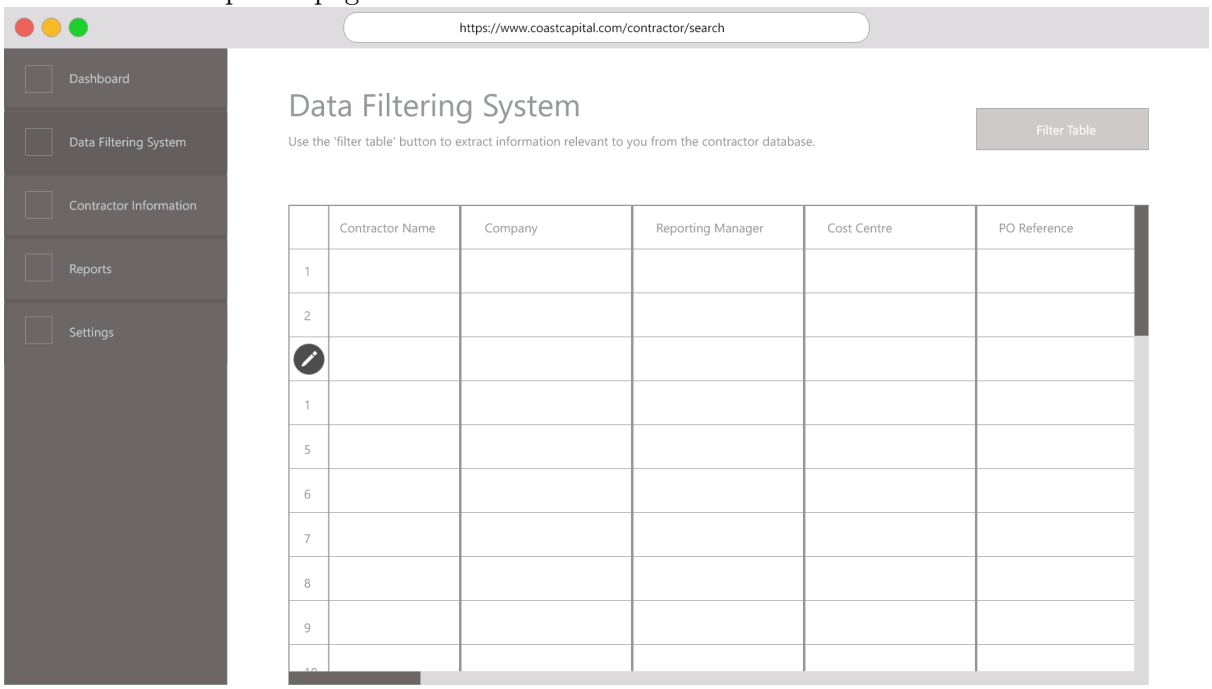
The wireframes in this document provides a horizontal wireframe with some vertical functionalities to show the system flow from the eyes of the user.



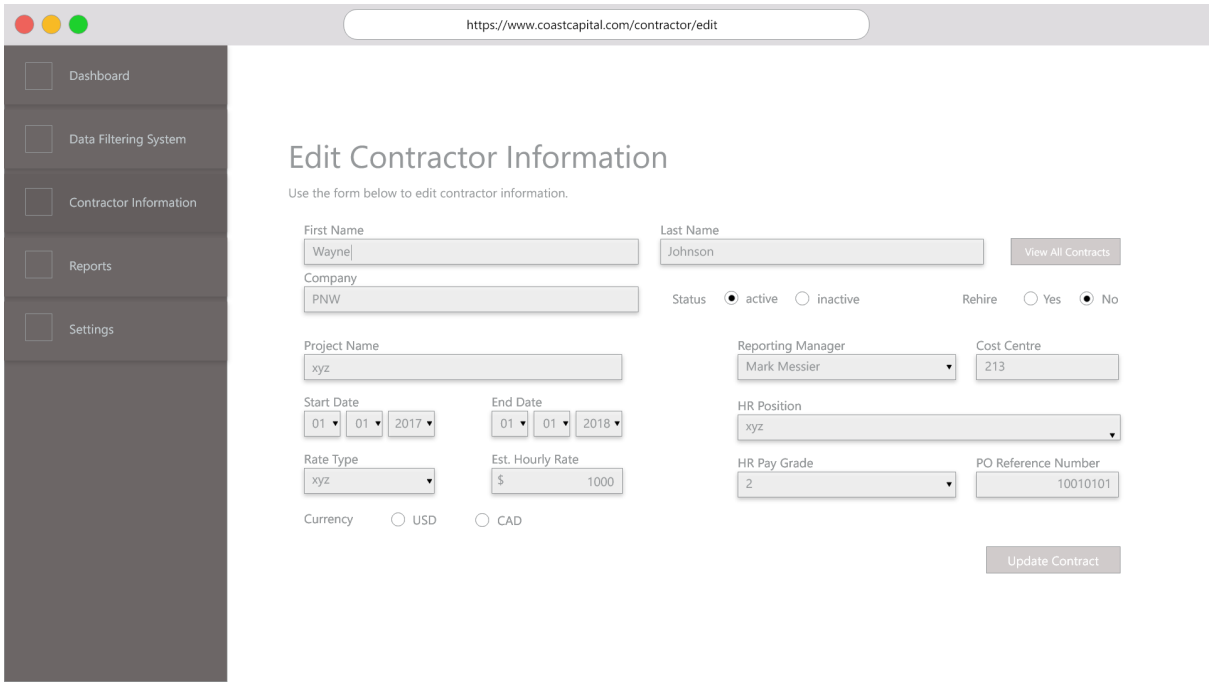
The login page is simply designed to allow us to know who is entering the system, and therefore what should be displayed as options once the user logs in.



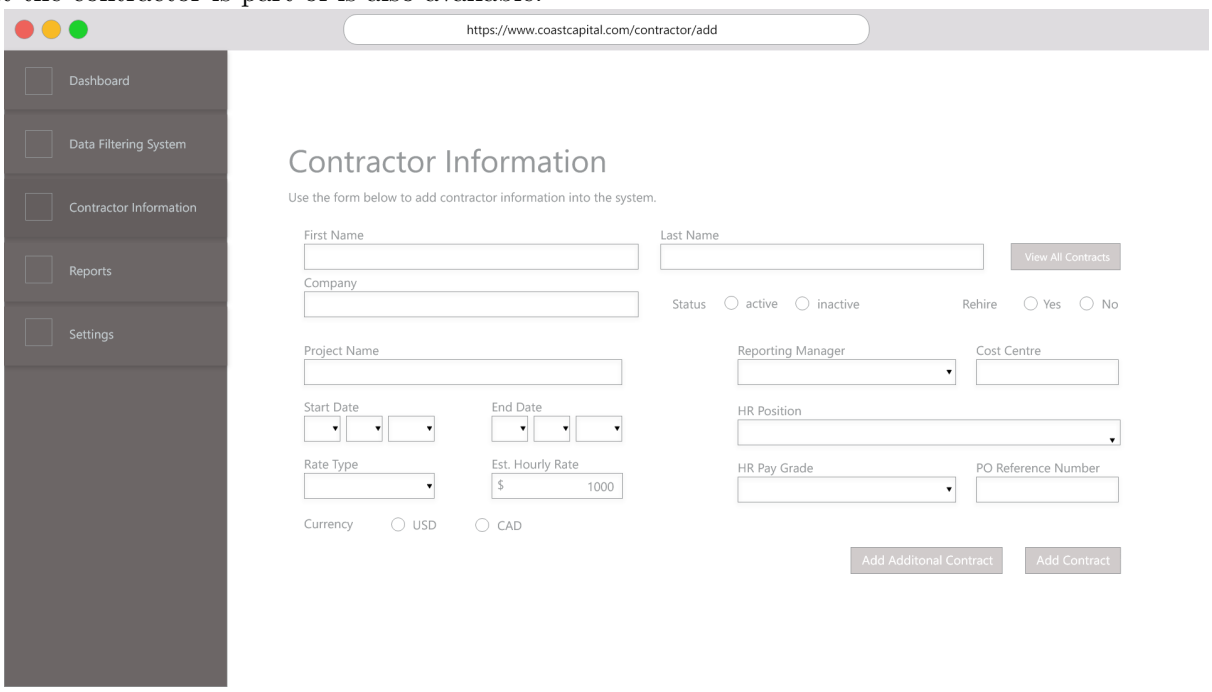
The dashboard allows the user to select the type of task they would like to do once logged in. While the same buttons are available on the left navigation panel, this allows the user to cleanly enter the web application, without the landing page overwhelming them with data that would otherwise be the data filtration page. Additional functionality (not shown) could be to have some form of a profile page.



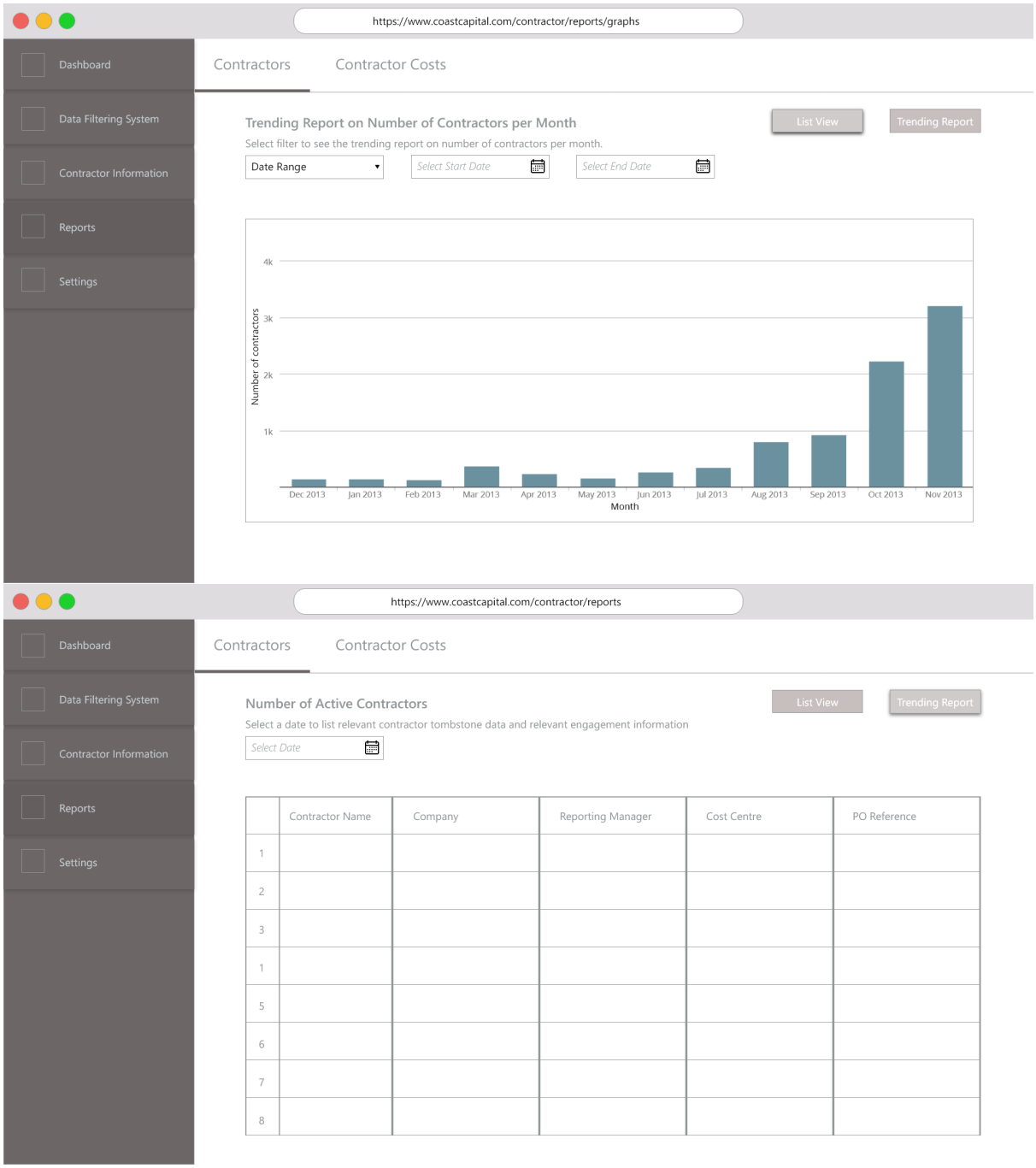
The data filtration system works a lot like the excel spreadsheet, by initially providing all the contractor data, until the user filters through, and selects what he or she would like to see. The similarity to the current system they have will provide good learnability, and efficiency in learning the system.



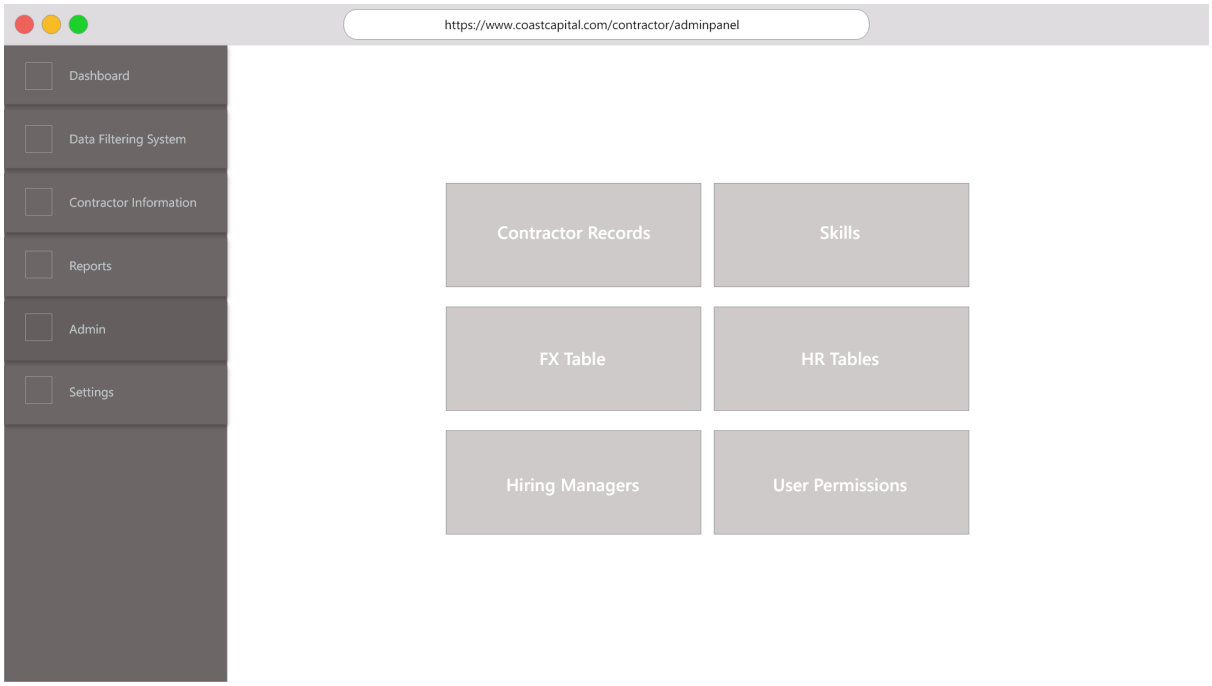
Users can directly choose to edit a particular tuple, which redirects them to a web-form style page where the fields are auto-populated with the data from the contractor table. Edits can be made to any field (but the field cannot be empty). The option to edit multiple contracts that the contractor is part of is also available.



The user can add contractors to the contractor database. The same web-form that we saw with edit contractors is made available, and there is the option to add multiple contracts for one contractor at one time. This prevents the system from being annoying and frustrating since multiple projects can be added at once.



Reports can be generated as lists or as bar graphs. The tab selection on top reduces the clutter, and makes the system very intuitive on what has been selected and what hasnt. Filtering on the graphs can be done through a drop down menu.



Admins have an additional dashboard that allows them to maintain various types of data tables.

3 Programming Environment

3.1 Summary

We will be using JavaScript, Java and MySQL for this project.

3.2 Front-End Technology

WebStorm	WebStorm provides great JS developing tools and its free for students.
JavaScript	The site is interactive, so we will need to use JS.
Node.js NPM	We will be using different libraries and packages, and NPM will help with managing them.
React	The site will have to be highly interactive and responsive, and React will provide that for us.
Redux	To achieve fast response times and avoid further API calls to the tables, Redux will provide a state management tool for us to store the data the first time we receive it.
webpack	This tool will be used for module bundling and integrating other tools and libraries (e.g. babel, linters, css loader, etc.)
Babel	Since we will be supporting IE11, it is essential for our JS code to be recompiled for compatibility. Babel will be doing that for us.
CSS Loader	This library will help us integrate CSS and JS.

3.3 Back-End Technology

IntelliJ	Intellij provides an IDE for developing in a wide variety of languages with excellent support for Java. It also is free for students.
DataGrip	DataGrip provides an IDE for developing databases, we will be using this for viewing, creating, and interacting with our SQL database (alongside interaction through Java code). DataGrip is provided free of charge for students.
Java	Java is our main programming language for the business logic of our application.
MySQL	We will be using MySQL as our relational database management system for storing data, aligning us with Coast Capitals requirements.
Amazon RDS	Amazon RDS (Relational Database Service) is where we will be hosting our MySQL database.
AWS Elastic Beanstalk	AWS (Amazon Web Services) Elastic Beanstalk is where we will be hosting our web application.
Spring	We will be using the Java Spring framework for creating REST endpoints as well as for its dependency injection features.

4 Prdouction and Test Environments

4.1 Front-End Technology

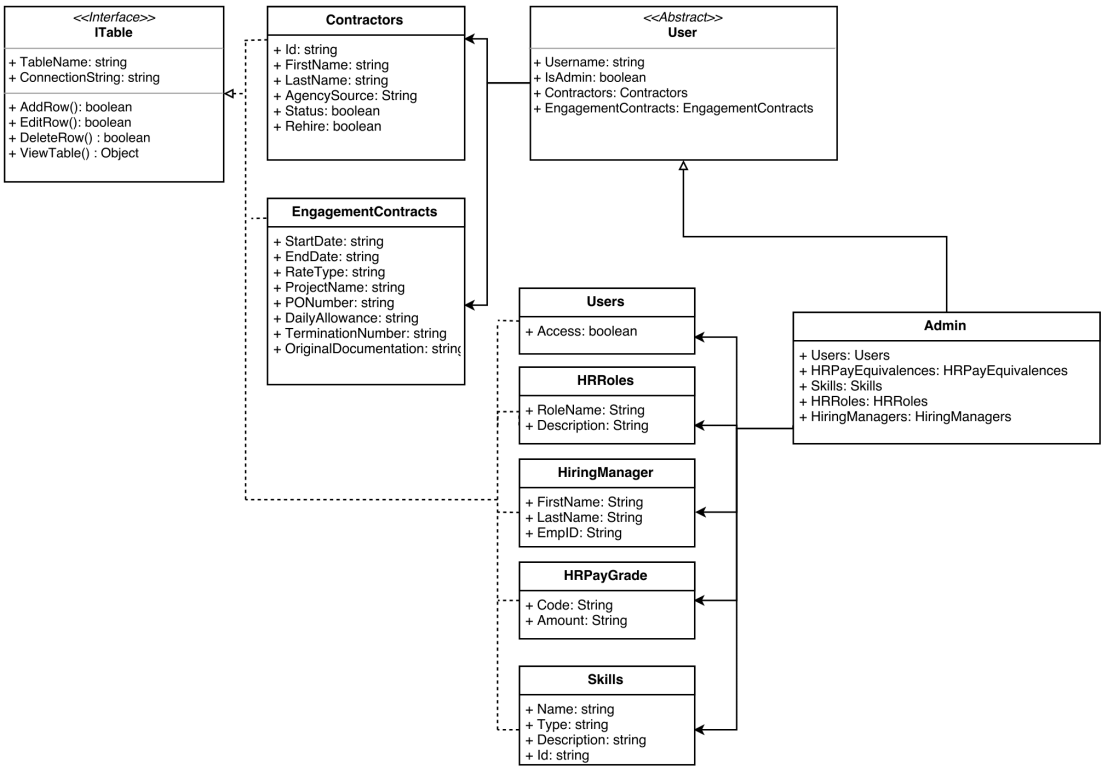
Selenium	Selenium is a tool that allows us to automate testing, eliminating the risk of forgetting to test through manual testing, and speeding up the process of an otherwise menial task. It also allows for frequent regression testing.
Mocha	Mocha is very useful for testing asynchronous calls in JavaScript. It uses node.js and is useful for mapping uncaught exceptions to the correct test cases.
Chai	Chai is a great extension for Mocha which will give us more in-depth unit testing abilities.
Nock	In order to test API calls from the front-end, Nock will give us a fake server, function spies, fake responses, etc.

4.2 Back-End Technology

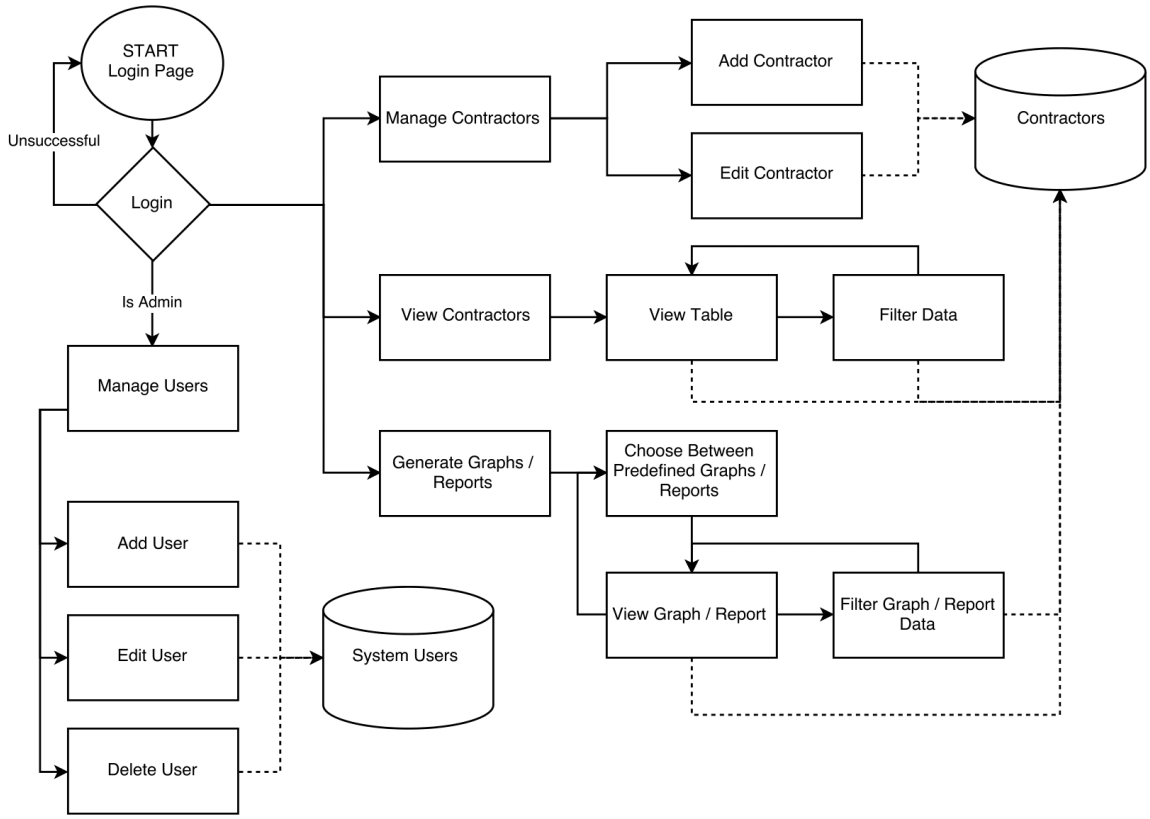
JUnit	We will be using JUnit for unit testing all of our Java code.
Mockito	We will be using Mockito to mock certain aspects of our application to allow for easier unit testing (such as mocking database connections).

5 Software Architecture

We visualized the back-end and front-end architecture using a UML Class Object and System Flow diagram. The UML diagram helps us to visualize the structure of the various classes and their attributes. We have all the different kinds of tables implementing the table interface since each of them has similar functionality even though they have different attributes. The admin class extends the user class and has extra functionalities for the admin specific tasks.



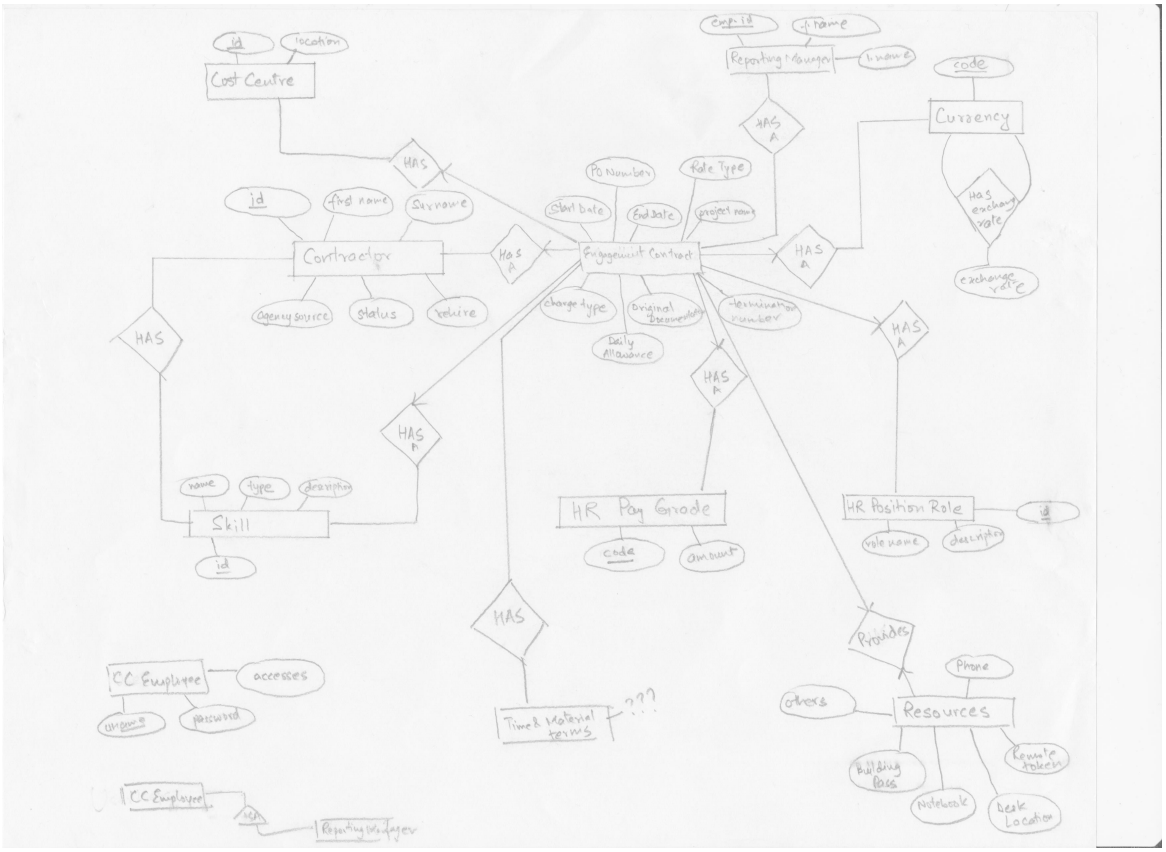
The system flow shows the communication between the frontend and backend as well as the structure of the frontend code. As shown in the diagram below, each user is validated while logging in which determines if they get administrative access. The administrator has the right to add, edit, or delete users. All users in the system can add and edit contractors, view and filter tables or generate reports/graphs. The admin's management of other tables isn't shown explicitly in the diagram but pls refer to the mock up screen



6 Data Design

6.1 ER Diagram

Our ER diagram showcases the data and relationships we will be capturing within our database. The ER diagram shows the cardinality of relationships between tuples and also highlights the primary keys as well as other attributes each tuple will contain. It is from this ER diagram that a normalized database schema was created.



6.2 Normalized Database Schema

Here is the list of tables:
(Undelrined attribute is the primary key whereas bolded attribute is a foreign key)

1. User (<u>username</u>, password, permissions)
2. Contractor(<u>id</u>, firstName, surname, agencySource, status, rehire)
3. EngagementContract(<u>id</u>, startDate, endDate, rateType, projectName, chargeType, dailyAllowance, originalDocumentation, terminationNum, **contractorId, resourceId, hrPositionId, hrPayGradeId, costCenterId, reportingManagerUserId, currencyCode, mainSkillId, materialNeeds**)
4. Skill(<u>id</u>, name, type, description)
5. SkillProvided(**<u>skillID</u>, <u>contractorID</u>**)
6. ResourceProvided(<u>id</u>, **engagementID**, phone, remoteToken, buildingPass, notebook, deskLoc, others)
7. HRPositionRole(<u>id</u>, roleName, description)
8. HRPayGrade(<u>code</u>, startAmount, endAmount)
9. CostCenter(<u>id</u>, location)
10. Currency(<u>code</u>, country)
11. FXRate(**<u>curCode1</u>, <u>curCode2</u>**, rate)

12. HiringManager(<u>userID</u>, firstName, lastName)

13. Login (**<u>userID</u>**, <u>token</u>)

The normalized database schema shows the tables required for capturing all of the information used by our application. The schema has primary keys underlined and foreign keys bolded. The database is normalized into third normal form to allow for fast access of necessary data and to reduce redundant data stored within the database.

7 API Design

7.1 REST API Calls

1. Login user
   Allows a user to login to the web application.
   (a) **URL** : /login
   (b) **Method** : GET
   (c) **URL Params** : username=[String], password=[String]

2. Logout user
   Allows a user to logout from the web application.
   (a) **URL** : /logout
   (b) **Method** : GET
   (c) **URL Params** : username=[String], token=[String]

3. Refresh User Validates that the user is already logged in.
   (a) **URL** : /refresh
   (b) **Method** : GET
   (c) **URL Params** : username=[String], password=[String]

4. Add User
   Allows an admin user to create a new user in the database.
   (a) **URL** : /users/add
   (b) **Method** : POST
   (c) **URL Params** : token=[String], username=[String], password=[String], permissions=[String]

5. Update User
   Allows an admin user to update an existing user in the database.
   (a) **URL** : /users/edit
   (b) **Method** : POST
   (c) **URL Params** : token=[String], username=[String], password=[String], permissions=[String]

6. Delete User
   Allows an admin user to delete an existing user from the database.
   (a) **URL** : /users/delete

(b) **Method** : POST

(c) **URL Params** : token=[String], username=[String], userToDelete=[String]

7. View Contractors
   Fetches tombstone data for all the contractors.

   (a) **URL** : /contractors/view

   (b) **Method** : GET

   (c) **URL Params** : token=[String]

8. View All Contractor Data
   Fetches data relating to all contractors (active and inactive) and their respective engagement contracts.

   (a) **URL** : /contractors/viewAllData

   (b) **Method** : GET

   (c) **URL Params** : token=[String]

9. View Contractor Data for Reports
   Fetches all the relevant data about all contractors required for generating reports and filtering

   (a) **URL** : /contractors/viewReportData

   (b) **Method** : GET

   (c) **URL Params** : token=[String]

10. Update Contractor
    Updates data relating to the given contractor in the database.

    (a) **URL** : /contractors/edit

    (b) **Method** : POST

    (c) **URL Params** : token=[String], id=[String], firstName=[String], surname=[String], agencySource = [String], status=[String]

11. Add Contractor
    Adds the given contractor to the database.

    (a) **URL** : /contractors/add

    (b) **Method** : POST

    (c) **URL Params** : token=[String], id=[String], firstName=[String], surname=[String], agencySource = [String], status=[String]

12. Update EngagementContract
    Updates data relating to the given engagement contract.

    (a) **URL** : /contractors/edit/engagementContract

    (b) **Method** : POST

    (c) **URL Params** : token=[String], id=[String], startDate=[String], endDate=[String], rateType=[String], projectName=[String], chargeType=[String], dailyAllowance=[int], originalDocumentation=[String], terminationNum=[int], contractorId=[String], resourceId=[String], hrPositionId=[String], hrPayGradeId=[String], costCenterId=[String], reportingManagerId=[String], currencyCode=[String], mainSkillId=[String], timeMaterialTerms=[int], poNum=[int], hourlyRate=[int]

13. Add EngagementContract
    Adds the given EngagementContract(s) to the database.
    (a) **URL** : /contractors/add/engagementContract
    (b) **Method** : POST
    (c) **URL Params** : token=[String], startDate=[String], endDate=[String], rateType=[String], projectName=[String], chargeType=[String], dailyAllowance=[int], originalDocumentation=[String], terminationNum=[int], contractorId=[String], resourceId=[String], hrPositionId=[String], hrPayGradeId=[String], costCenterId=[String], reportingManagerId=[String], currencyCode=[String], mainSkillId=[String], timeMaterialTerms=[int], poNum=[int], hourlyRate=[int]

14. Update Skill
    Updates data relating to the given skill in the database.
    (a) **URL** : /skills/edit
    (b) **Method** : POST
    (c) **URL Params** : token=[String], id=[String], name=[String], description=[String], type=[String]

15. Add Skill
    Adds the given skill to the database.
    (a) **URL** : /skills/edit
    (b) **Method** : POST
    (c) **URL Params** : token=[String], name=[String], description=[String], type=[String]

16. View Skills
    Fetches data for all the Skills
    (a) **URL** : /skills/view
    (b) **Method** : GET
    (c) **URL Params** : token=[String]

17. Update HR PayGrade(HRPayGrade)
    Updates the given HRPayGrade in the database
    (a) **URL** : /paygrades/edit
    (b) **Method** : POST
    (c) **URL Params** : token=[String], id=[String], startAmount=[int], endAmount=[int], name=[String]

18. Add HR PayGrade
    Adds the given HRPayGrade to the database.
    (a) **URL** : /paygrades/add
    (b) **Method** : POST
    (c) **URL Params** : token=[String], startAmount=[int], endAmount=[int], name=[String]

19. View PayGrades
    Fetches all paygrades data
    (a) **URL** : /paygrades/view

(b) **Method** : GET

(c) **URL Params** : token=[String]

20. Update HR Role
    Updates the given HRRole in the database.

    (a) **URL** : /hrroles/edit

    (b) **Method** : POST

    (c) **URL Params** : token=[String], id=[String], roleName=[String], description=[String]

21. Add HR Role
    Adds the given HRRole to the database.

    (a) **URL** : /hrroles/add

    (b) **Method** : POST

    (c) **URL Params** : token=[String], roleName=[String], description=[String]

22. View HR Roles
    Fetches all hr roles data

    (a) **URL** : /hrroles/view

    (b) **Method** : GET

    (c) **URL Params** : token=[String]

23. Update HiringManager
    Updates the given HiringManager in the database.

    (a) **URL** : /hiringmanagers/edit

    (b) **Method** : POST

    (c) **URL Params** : token=[String], id=[String], firstName=[String], lastName=[String]

24. Add Hiring Manager
    Adds the given HiringManager(s) to the database.

    (a) **URL** : /hiringmanagers/add

    (b) **Method** : POST

    (c) **URL Params** : token=[String], firstName=[String], lastName=[String]

25. View Hiring Managers
    Fetches tombstone data for all the contractors.

    (a) **URL** : /hiringmanagers/view

    (b) **Method** : GET

    (c) **URL Params** : token=[String]

26. View Cost Centers
    Fetches all cost centers

    (a) **URL** : /costcenters/view

    (b) **Method** : GET

    (c) **URL Params** : token=[String]

27. View FX Rates
    Fetches all fxrates

    (a) **URL** : /fxrates/view

    (b) **Method** : GET

    (c) **URL Params** : token=[String]

7.2 API Calls-External

We will be calling the fixer.io API to get the latest currency exchange rates so that a user does not have to update the currency exchange rates manually. Specifically, we will be using a GET call to http://api.fixer.io/latest?base=CAD&symbols=USD to get the exchange rate between the Canadian dollar and the United States dollar. This API will be automatically called once per day to update the FXRate table to the latest exchange rate.

8 Algorithms

We will not be using any complex algorithms in our code. One of the more complicated processes will involve implementing a state management environment (Redux) for front-end. The state will be created once a session starts, and it will feed each React component the data it needs to render. When the user interacts with the page, actions will be dispatched to update the state of the app accordingly. These interactions can be as small as typing something into a form or as big as requesting data from the back-end.

9 Notable Tradeoffs

1. Since we are using AWS for our hosting needs opposed to setting up our own physical server infrastructure, we have less control over the actual hardware used to host our server. However, setting up our server on AWS is a lot faster and easier than setting up our own infrastructure and AWS already has scaling built in so we do not have to worry about those issues. Furthermore, we do not have to deal with physical server maintenance as Amazon takes care of that for us.

2. When filtering tables/graphs, the client will take care of it instead of calling an API to ask the back-end to do it. It will be faster and easier to manage because there will be fewer API calls. However, the front-end code will be more complicated and if the data is very big, filtering can be slow.

3. When filtering tables/graphs, the client will take care of it instead of calling an API to ask the back-end to do it. It will be faster and easier to manage because there will be fewer API calls. However, the front-end code will be more complicated and if the data is very big, filtering can be slow.

10 Notable Risks

1. One notable risk is using AWS to host our database and web application, if AWS were to go down then our web application will not function. However, AWS provides a guarantee of 99.95% Monthly Uptime Percentage so the likelihood that AWS will go down is very low.

2. Another risk is if we do not implement safety prompts or an automatic session expiry for the user. Implementing safety prompts in the UI would ensure that the user does not accidentally change data and an automatic session expiry would ensure that the user does not expose sensitive data.