7Factor Staffing Tool

A Major Qualifying Project Report

Submitted to the faculty of Worcester Polytechnic Institute in partial fulfillment of the requirements for the Degree of Bachelor of Science

In cooperation with 7Factor, LLC

Submitted May 18, 2021

By:

Panagiotis Argyrakis

Rafael Pimentel

Nicholas J. Wood

Project Advisor:

Professor Joshua Cuneo

Abstract

7Factor Software is a consulting company that completes projects for clients on a contract basis. They need to track the time each employee spends on each project so they can bill their clients correctly. They also track their staff utilization so they can book the correct number of new contracts to ensure every employee is working on paid tasks. In this project we present how we designed a web platform to fulfill those needs and implemented several of the back-end and front-end components. We also describe the future work needed to complete the application.

Acknowledgements

We would like to thank the following people for their involvement and help throughout our project:

Professor Joshua Cuneo, project advisor

Jeremy Duvall, CEO of 7Factor

Kenneth Jørgensen, 7Factor software developer

Chelsea Green, 7Factor software developer

This project would not have been possible without their help, expertise and guidance.

List of Figures

Executive Summary

About 7Factor

7Factor is a software consulting company based in Atlanta, Georgia. They operate on a contract basis and develop software for clients in various locations. They specialize in creating, deploying and advising on custom dev-ops and cloud-based solutions for clients of sizes ranging from startups to large corporations. The 7Factor team members work remotely from locations around the world.

Time Tracking Needs and Existing Solution

7Factor bills clients based on the engineering time spent on their projects. Each consultant is billed at a different hourly rate that depends on the skill set of the consultant, the type of client and the characteristics of the project itself. The same engineer could be billed to different projects at different rates.

To fulfill their time tracking needs, the company uses a Software-as-a-Service platform called TimeIQ. It provides sufficient functionality but has two major disadvantages. First, the user interface is overly complicated which creates a frustrating user experience both for the platform administrator and the employees who log time. Second, it does not support customizing the billing rate for each engineer/project pairing individually. Instead, the administrator has to use a time-consuming workaround.

The goal of our project was to design a new web platform to replace TimeIQ. This platform should carry over many of the desired features of its predecessor, but be much simpler to use and support easily assigning a custom rate to an engineer.

Design of the New Solution

In our design, we maintained the concept of a project, client, employee and service (i.e. rate card), but added an option to use an hourly rate that is not on the project rate card when the administrator creates or edits a consultant/project pairing. We carried over the summary graphs of total revenue versus total expenses for each project. We used a very simple, Material Design-inspired theme and avoided many of the redundant views that TimeIQ provides. Instead, the app defaults to the time entry screen where employees can enter their weekly time logs for every project they are assigned to.

To create the web platform, we used Postgres DB for the database, the Spring Boot Java framework for the REST API and ReactJS for the front-end. We followed the model-view-controller architecture. We organized our team by following the SCRUM software development methodology and held weekly sprint planning meetings and daily stand-up meetings.

Results

Within the timeframe of our project, we designed and implemented the database, completed the majority of the back-end, created complete User Interface mock-ups and wrote a basic front-end in React. We also implemented basic JWT authentication through the Auth0 provider. We spent much of our time implementing the database and back-end which support a significantly larger portion of the functionality than the front-end.

Future Work

Several additions are needed before the platform can be deployed. First, the front-end needs support for the editing of rate card, employee and project entities all of which are implemented in the back-end. More advanced object validation, error handling and user feedback when errors occur would significantly improve the reliability and user experience.

Another feature addition is the ability to hide which projects an employee views when entering time, based on which ones they are actively working on at the moment. This would allow for a better experience for the user since there would not be as many time entry rows visible on the screen. In addition, the ability to set a default time for new time entries would further improve the ease of use since many time entries follow the same pattern.

Integrating both the back-end and the front-end with the Continuous Integration pipeline of 7Factor will also improve the quality of the application and speed up development. Furthermore, the UI would benefit from a styling overhaul and implementing dynamic resizing.

Modifying the database, back-end and front-end to support a `Clients` table is another needed improvement. In our implementation, a client is represented only as a free text field in each project entity. As a result, it is difficult to track which clients have more than one active project and calculate metrics across all projects of a specific client.

A technical issue we faced was loading lazily-fetched objects in the front-end. In the current implementation, the front-end often fetches objects along with any nested object collections within them. Modifying the back-end and front-end to support lazily loading nested assets would improve the performance of the application. A different, more thorough approach of solving this issue would be to construct purpose-built Data Transfer Objects (DTOs) that only include the information that is strictly necessary to facilitate different controller calls. DTOs would also drastically increase the security of the application, protect objects against accidental changes and reduce the need for validating every object field at the controller level.

Last, adding a search feature in various parts of the interface, such as when viewing all employees or projects would improve the usability of the interface.

Introduction

7Factor is a software company based in Atlanta, Georgia. They operate on a contract basis and develop software for clients in various locations. They specialize in creating, deploying and advising on custom dev-ops and cloud-based solutions for clients of sizes ranging from startups to large corporations. The 7Factor team members work remotely from locations around the world.

The goal of this Major Qualifying Project (MQP) was to build a Software-as-a-Service (SaaS) platform that combines time tracking and accounting features and allows 7Factor to record and track information about how much time employees spend on different projects and how much revenue they generate. 7Factor intends to use this system to log employee time entries, but also calculate more complex information such as billing statements for clients, staff utilization, and profitability.

There are complexities associated with the way 7Factor bills clients. The hourly rate at which each engineer is billed to a client depends on the skill set of the engineer, the type of client and the characteristics of the project itself. So, the same engineer could be billed to different projects at different rates. To calculate all the needed metrics and statements reliably, the platform needs to record the rate at which the specific engineer/project pairing is billed to clients. Moreover, it needs to provide an interface where employees can record the time they spent on each of the projects they are assigned to. To calculate the profitability of the business, the platform must record the fixed yearly salary of each employee and compare it with the revenue generated. These are some of the basic challenges and requirements of the platform.

The new platform will replace TimeIQ, the existing SaaS platform 7Factor uses. TimeIQ provides sufficient functionality for their needs but has two major disadvantages that the new system is designed to overcome. First, the user interface is overly complicated which produces a frustrating user experience both for the platform administrator and the employees who log time. Second, TimeIQ does not support customizing the billing rate for each engineer/project pairing individually. Instead, the administrator has to go through the time-consuming process of creating a whole new “service” entity. The new platform aims to address those issues and create a faster and easier workflow for 7Factor staff.

Due to the large scope and complexity of developing this SaaS platform, our goal was not to deliver the finished product but rather define the design of the system and implement its core components. We completed a significant part of the back-end and database stack and created a basic front-end in React that can be extended in the future. In this paper, we will describe the process that we followed to design and implement the new web platform, the challenges we faced, our final deliverable and what further work is needed before the platform can be deployed.

Background

Employee Cost and Timecard Tracking Needs

7Factor employees are paid a fixed salary. They track the time they spend on different projects and time off. If there are not enough ongoing projects to occupy all employees, 7Factor uses the concept of the “bench” to record time spent not working on paid projects. Hours on the “bench” do not affect the salary of the employee but affect the profitability of the company.

The company bills each project to clients based on the total development time spent on it. The hourly rate that the client pays for each engineer depends on the characteristics of the project, the client, and the specific skill set of the engineer. The same engineer may be billed at a different rate at each different project. Specifically, the invoice a client receives is described by:

$$\sum_{e\,=\,employee}^{n_e} \sum_{d\,=\,project\ start\ date}^{project\ end\ date} timeSpent(e,\ d,\ p)\ *\ hourlyRate(e,\ p)$$

where p is the project that is being billed and n_e is the total number of employees.

The expression above shows that the *hourlyRate* depends on the project and employee, and the *timeSpent* on the project depends on the employee, date, and project. It also shows that to calculate the final invoice for a project, we need to sum the time entries of all engineers for all dates they worked on the project and multiply the total time each engineer spent on the project with their project-specific hourly rate.

7Factor intends to use the time tracking system to make invoice calculations like the one above, track overall profitability and measure the availability of each employee. Maintaining a reliable measure of staff utilization is crucial because the company can use it to gauge how much extra work they can take on and avoid overbooking or underbooking their staff.

Overview of the Existing System - TimeIQ

7Factor currently uses a web-based time tracking platform called TimeIQ. TimeIQ is a SaaS product that provides time tracking functionality for employees along with various accounting and administrative tasks for business managers. Within the platform, an administrator can create services, clients and projects, among other entities. Each service consists of several rates. For example, the “Enterprise” service may specify the hourly rates for a “Senior Engineer” and a “Junior Engineer”. When an employee is assigned to work on a project, the administrator has to associate one of the aforementioned rates with that employee. The client will pay for engineering time according to that rate. If the administrator wants to use a rate that does not already exist in a service, they have to create a new service that includes the desired rate.

TimeIQ also provides several additional features such as a graph that shows the total revenue versus the total expenses of the business. It achieves this because the administrator has the option to enter salary information about the employees which the software uses to produce these summary graphs. These features are useful because they provide a top-down view of the business and help define and track profitability goals.

Issues with the Existing System

TimeIQ is a very feature-rich platform but fails in two significant ways. First, it does not provide a quick way to select a custom rate when assigning an employee to a project. Because services are tightly coupled to clients and projects, the administrator has to create a new service when they need to use a different rate for a particular engineer/project pairing. This process is very time-consuming for 7Factor and introduces significant administrative overhead.

The second major issue with TimeIQ is that it fails to tame its complexity. For example, it provides employees with several different ways to enter time, one way being to start and stop a timer, another being to manually enter the time in columns per day for a whole week. Even though the versatility is well-intentioned, it degrades the user experience for 7Factor staff because they have to go through several menus and options before finding the features they most frequently use.

Requirements of the New System

As a result of the high complexity and poor support for custom rates in TimeIQ, 7Factor wants a new system that fits their workflow better. The new platform must facilitate time tracking of each consultant to each project they are assigned to. It should be less complex and easier to use, and support quickly assigning employees to projects with a custom, project-specific hourly rate. If time allows, 7Factor also wants to see visualizations of revenue vs. expenses and a representation of overhead costs, such as the cost incurred when a consultant is not working on a paid project but still needs to work. The requirements are expressed in the following user stories:

Core requirements:

1. As an administrator, I want to invite new consultants to the platform so they can create accounts.
2. As an administrator, I want to create a project along with its client, start date, end date and set of default billing rates.
3. As an administrator, I want to assign a consultant to a project along with the hourly rate at which they will be billed to the client.
4. As an administrator, I want to remove a consultant from a project.

5. As an administrator, I want to view the total billable hours on a project for a specific time range so I can bill the client.
6. As an administrator, I want to set the yearly salary of a consultant and specify which date the change will take effect on.
7. As an administrator, I want to correct an existing time entry on behalf of a consultant.
8. As a consultant, I want to log the hours I spent on each project, on overheads or on the bench for a specific week.
9. As a consultant, I want to request to modify an existing time entry so I can correct a mistake on a time entry that has already been approved.

Stretch requirements:

1. As an administrator, I want to see a graph of the total expenses versus the revenue of the business so I can better understand the profitability.
2. As an administrator, I want to see a graph of the expenses versus the revenue for a specific project so I can better understand the profitability of a project.
3. As an administrator, I want to archive a finished project.
4. As an administrator, I want to view archived projects.

Methodology

Overview of Methodology

We started the design process by holding an initial meeting with 7Factor, during which we asked questions about the detailed requirements of the system. We were interested to know in detail what worked well and what did not with the previous system so we could replicate the desired features and redesign the rest. After the initial meeting, we created user stories and task sequences for the various features that the new platform would enable. Specifically, the user stories allowed us to understand exactly what the final goal was from the perspectives of each class of user, namely the employee and the administrator. The task sequence diagrams allowed us to view the steps involved in performing the actions described by the user stories and optimize the path towards the end goal. After completing these steps, we received feedback from the 7Factor team.

After completing documenting the requirements, we started designing the user interface. We first designed basic wireframes to illustrate the structure and flow of each page. After receiving feedback, we created graphical mockups to illustrate the styling. Once the user interface was designed, we defined the structure of the database and the back-end of the application. The database was a crucial part of the process because it determined how information would flow across the application.

Once the user interface and the database were defined, we started our development sprints. The goal of each sprint was to complete a related set of features. It would start with a planning meeting where we would prioritize the features that were most important and would synchronize back-end with front-end work. During the sprint we held daily SCRUM meetings that lasted about fifteen minutes. Throughout the duration of the sprint we received feedback from each other and 7Factor by posting pull requests. We also scheduled demonstration meetings about once per month during the latter half of our project to receive feedback from 7Factor directly.

Interface Design

We designed the interface in three stages. First, we created wireframes. The wireframes were very easy to build prototypes that showcase the intended functionality and layout of an interface without final styling elements. To create them, we used the online tool Balsamiq. Each one of us picked three screens to design and was given creative freedom to add helper screens to support the core functionality. We then held a meeting when each team member presented their own version of how the application would “flow” and how different functions would be accessed and performed. We discussed the different variants and decided which one was the most intuitive, easy to use, and future-proof and could be implemented within our project timeframe. To finalize the screens that were designed by only one member, we gave each other feedback and made tweaks until all of us were satisfied.

We used wireframes to quickly create a prototype of our interface and get feedback from 7Factor during a demo session. After receiving feedback, we tweaked our wireframes where necessary and used them as a basis to create mockups. For the mockups, we followed the same method as when creating the wireframes. They were more labor-intensive interface prototypes that mimic the look and feel of the finished product. They allowed us to get feedback about the styling and graphic design aspects of our interface and also provided us with some graphical assets that we could use in our product. We went through two iterations of feedback and tweaks on the mockups before beginning to write the code for our application.

Wireframes and Resulting Mockups

In this section we present the initial wireframes and resulting mockups for every screen of our application. Within each screen, we will also provide a task sequence. The task sequence represents the expected steps a user would go through to complete a given goal on that specific screen. It also visualizes the user interactions and allows us to notice potential flow issues and optimize the user experience.

Enter Time Screen

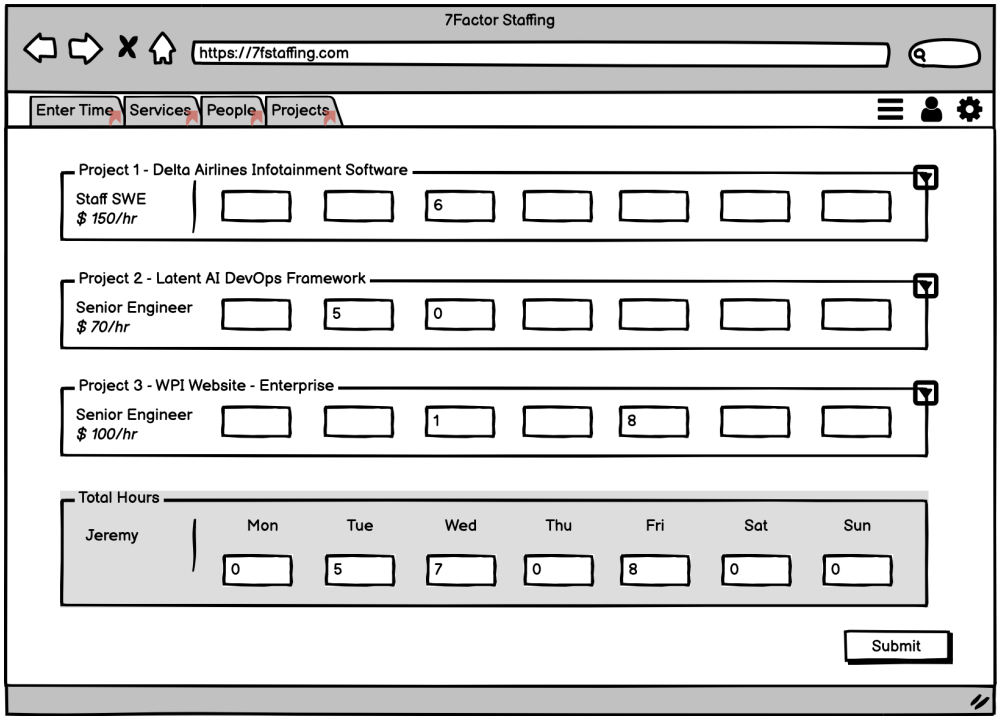

Figure 1: Our initial wireframe mockup for what we wanted time entry screen to look like

The enter time screen allows the user to submit time entries for any of the active projects the user is assigned to on a specific week (see figure 1). The interface also fetches any existing time entries and displays them. The user can select which week is displayed. It satisfies core requirements 8 and 9.

Task sequence for entering time: Log in → Go to Enter Time page → (Optional - defaults to current week) Select the week to enter time for → Fill in the time → Press submit

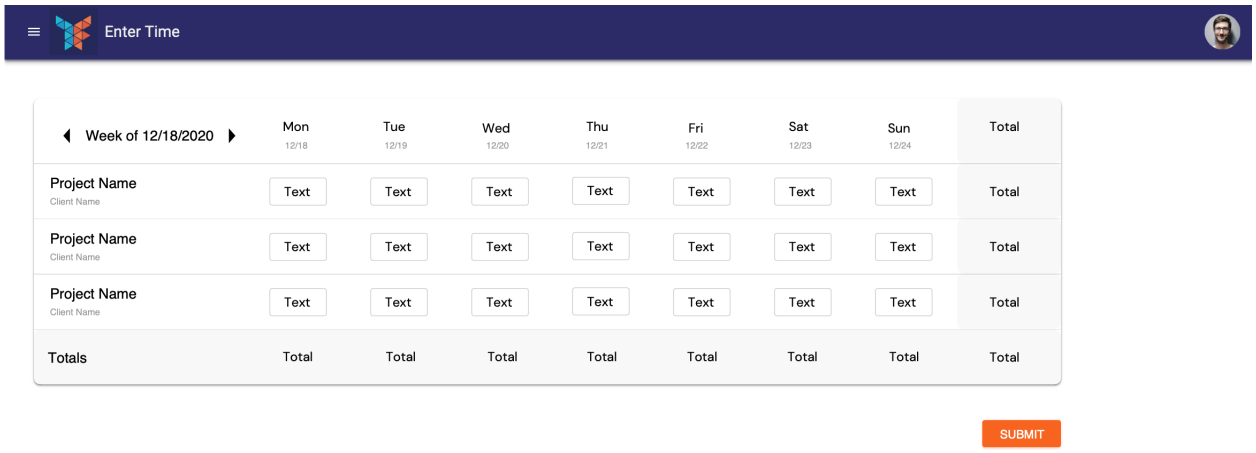

Figure 2: Updated mockup of time entry

In our final mockup as shown in figure 2, we edited the Enter Time page to include totals for both days and projects. We also added buttons to change the applicable date and added fine print under each day label to show the exact date.

People

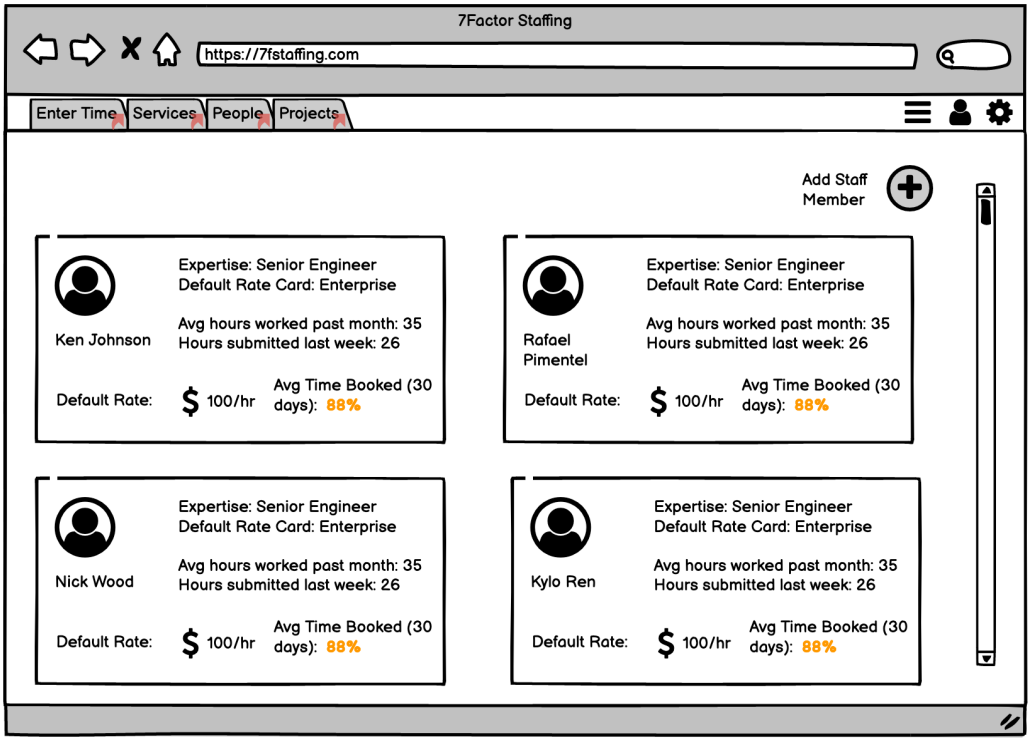

Figure 3: Our initial wireframe mockup for what the employee screen would look like

This screen, figure 3, displays all the employees of a company. It allows the administrator to add new employees to satisfy core requirement 1 and edit the information (such as the salary) of existing ones to satisfy core requirement 6.

Task sequences:

1. View all employees: Log in → Go to People page
2. Add a new employee:
    a. Log in → Go to People page → Click the plus button → Fill in the details for that person → Click add
    b. Alternate flows proposed:
    c. Admin log in → add employee email → employee receives form via email and fills in their details → admin fills in the rest of the info and confirms the registration
    d. Admin log in → add employee email and other info → employee receives form via email and fills in their details → admin confirms the registration

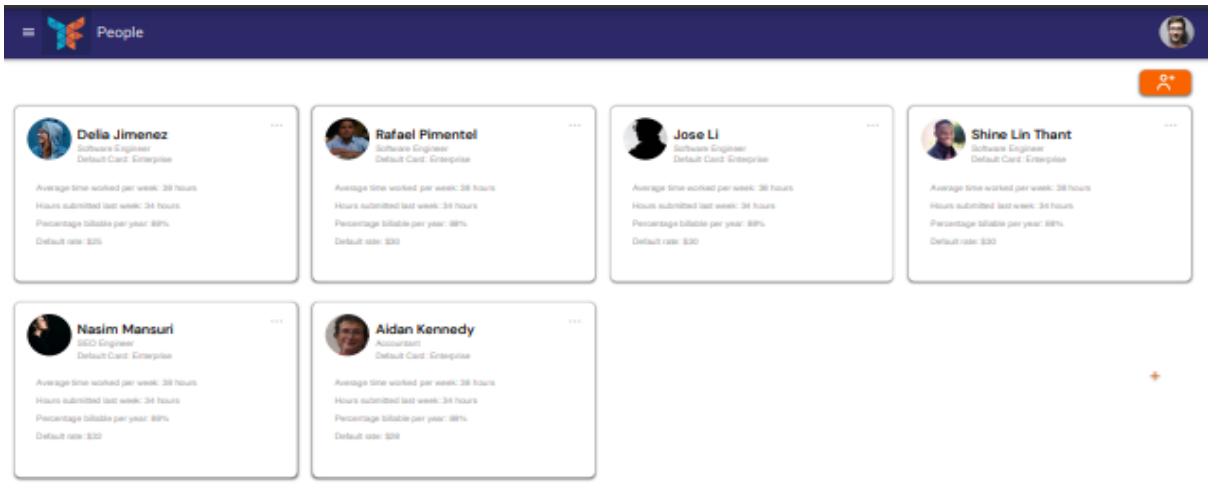

Figure 4: The updated employee screen

Our final mockup shown in figure 4 largely retained the same layout and look of the wireframe. We used a material design template to create consistent and good looking cards to represent employees.

Figure 5: Updated add employee screen

Our mockups also included an “Add a Person” screen, shown in figure 5, to specify how the form to create an employee would look like.

Projects

Figure 6: Wireframe of the overall project screen

The projects page shows all currently active projects along with basic information about them. It also shows an overview of revenue across all projects versus all expenses to provide an overview of the profitability of the business. This functionality satisfies stretch goal 1.

Task sequence for viewing all projects:

Log in → Go to Projects page → (optional) click next page if the project does not appear

Alternate flow:

Log in → Go to Projects page → (optional) type project name in the search bar at the top of the table

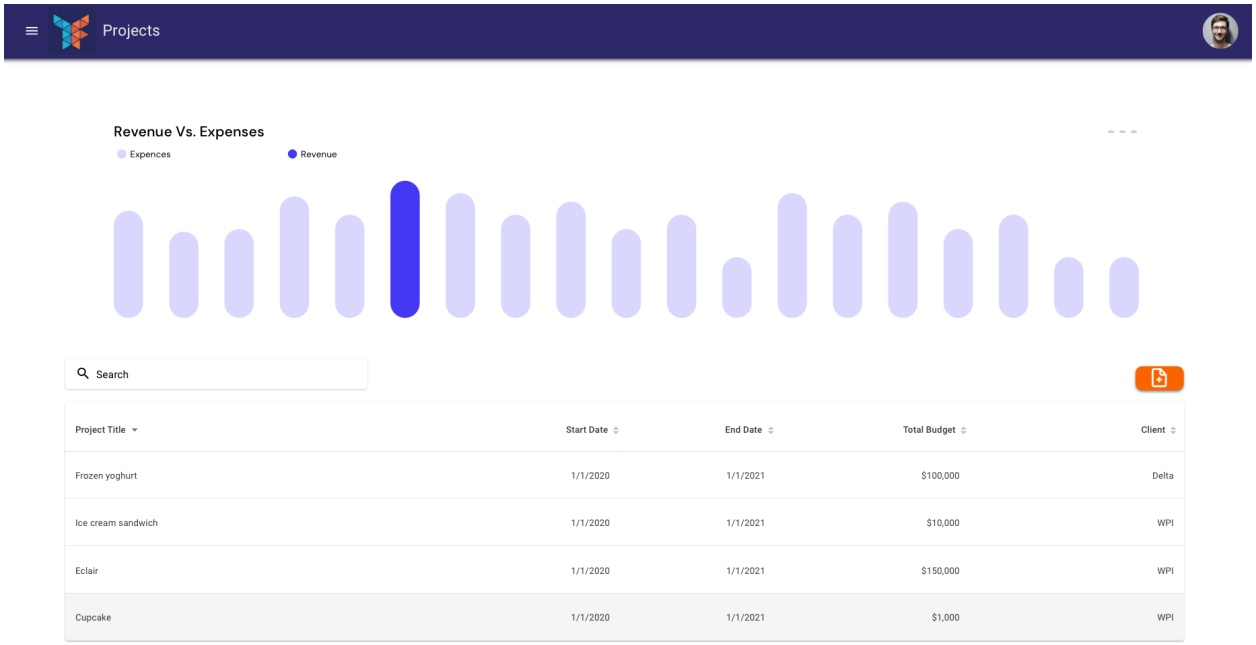

Figure 7: Long term stretch goal of revenue management screen

The most significant change in the Projects screen mockup, figure 7, is using a table to represent projects instead of cards. We learned that much of the information we were planning to display was redundant so a table would be a more digestible, usable and easy-to-develop solution.

Create/Edit a Project

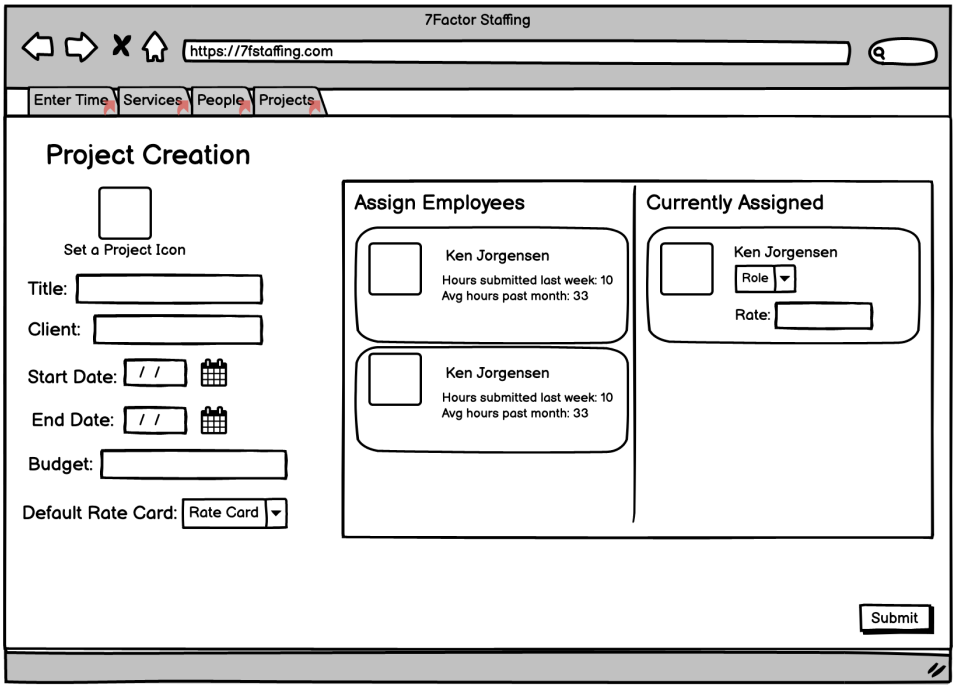

Figure 8: Wireframe mockup of project creation and employee assignment

The wireframe of creating/editing a project, figure 8, provides a form through which an administrator can create a project. They can set basic information such as the title, budget and key dates, and also assign employees to work on it. When assigning an employee, the administrator can specify a custom rate at which the consultant will be billed to the client. If left blank, they will be billed by looking up the role (i.e. job title) of the consultant in the default rate card of the project. These functions satisfy core requirements 2, 3 and 4.

Task sequences:

1. Create a project: Log in → Go to Projects page → Click the plus icon → fill in project information form → (optional) search for employees to add → (optional - defaults to no employees) drag and drop employees → (optional) set custom rate for employee → click submit

2. Edit a project: Log in → Go to Projects page → Go to project details (click on project) → Click the three dots button of a project → click edit → edit the project information form → (optional - defaults to no employees) drag and drop employees → (optional) set custom rate for employee → click save

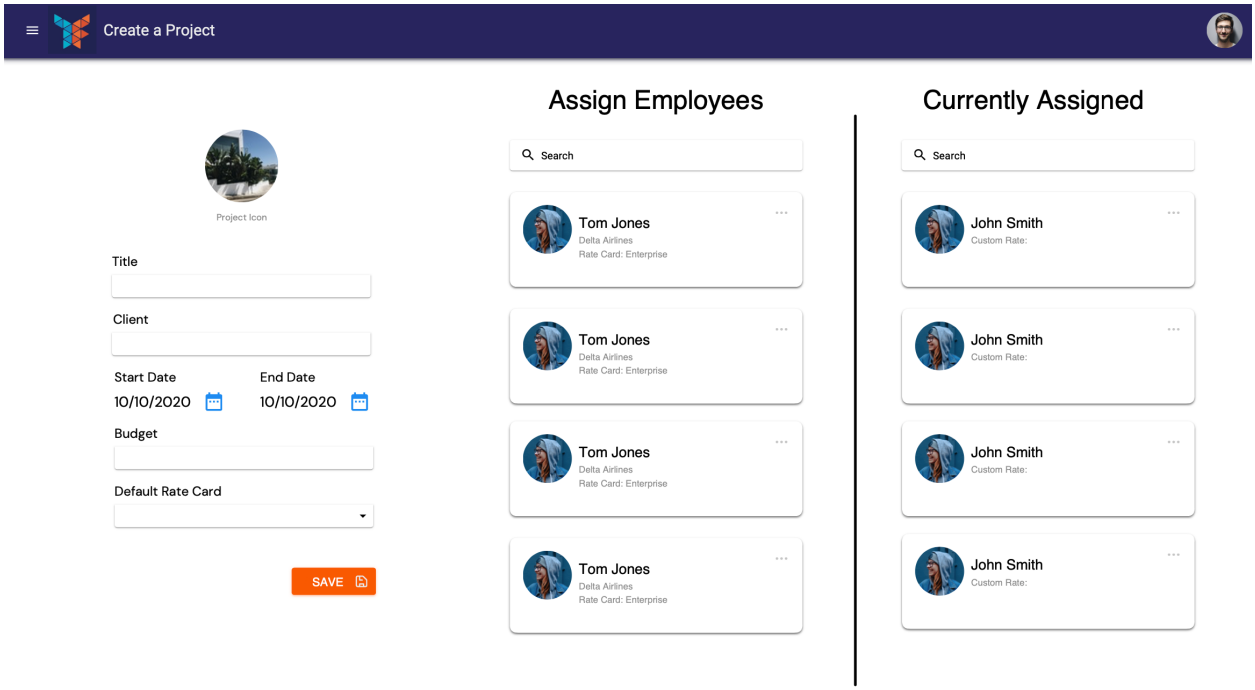

Figure 9: Updated mockup of create a project and assign employee

The design of the final mockup shown in figure 9 closely follows that of the wireframe.

Project Detail

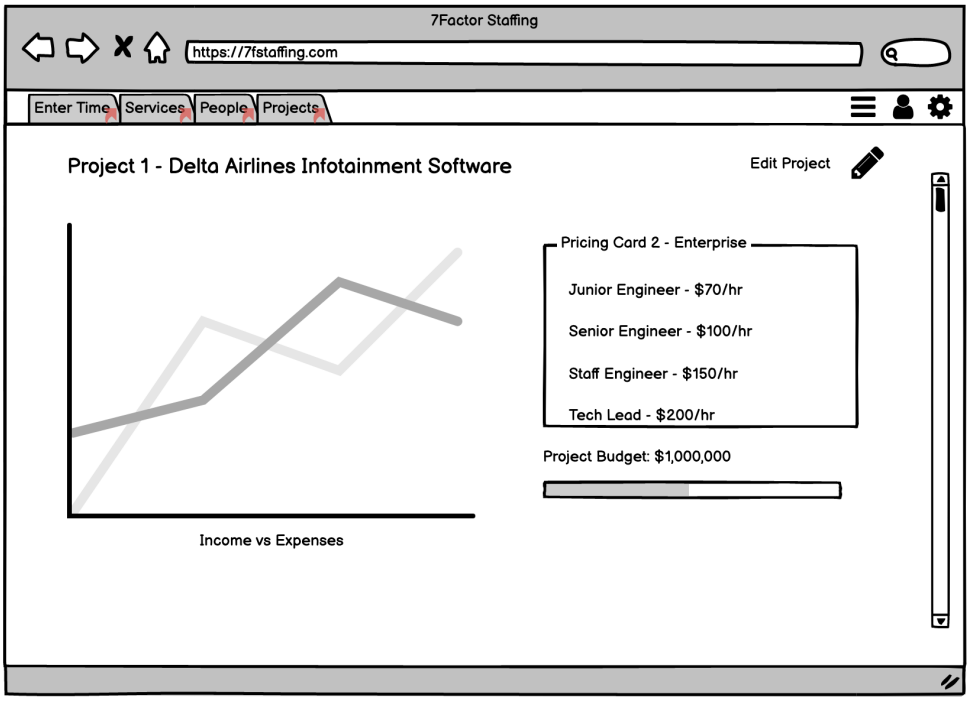

Figure 10: Wireframe of a project screen where we can see the details of a given project.

The administrator can click on a project in the View Projects screen and view details about that specific project (see figure 10). These include the assigned rate card, staff, budget, and dates. The screen also includes a graph of the income versus the expenses of the project so the admin can assess the profitability of it. The latter satisfies stretch requirement 2. Through this screen the administrator can edit the project. The “Edit project” screen is not shown because it is identical to the “Project Creation” screen.

Task sequence for viewing project detail:

Log in → Go to Projects page → Click on a project

Project Detail - Continued

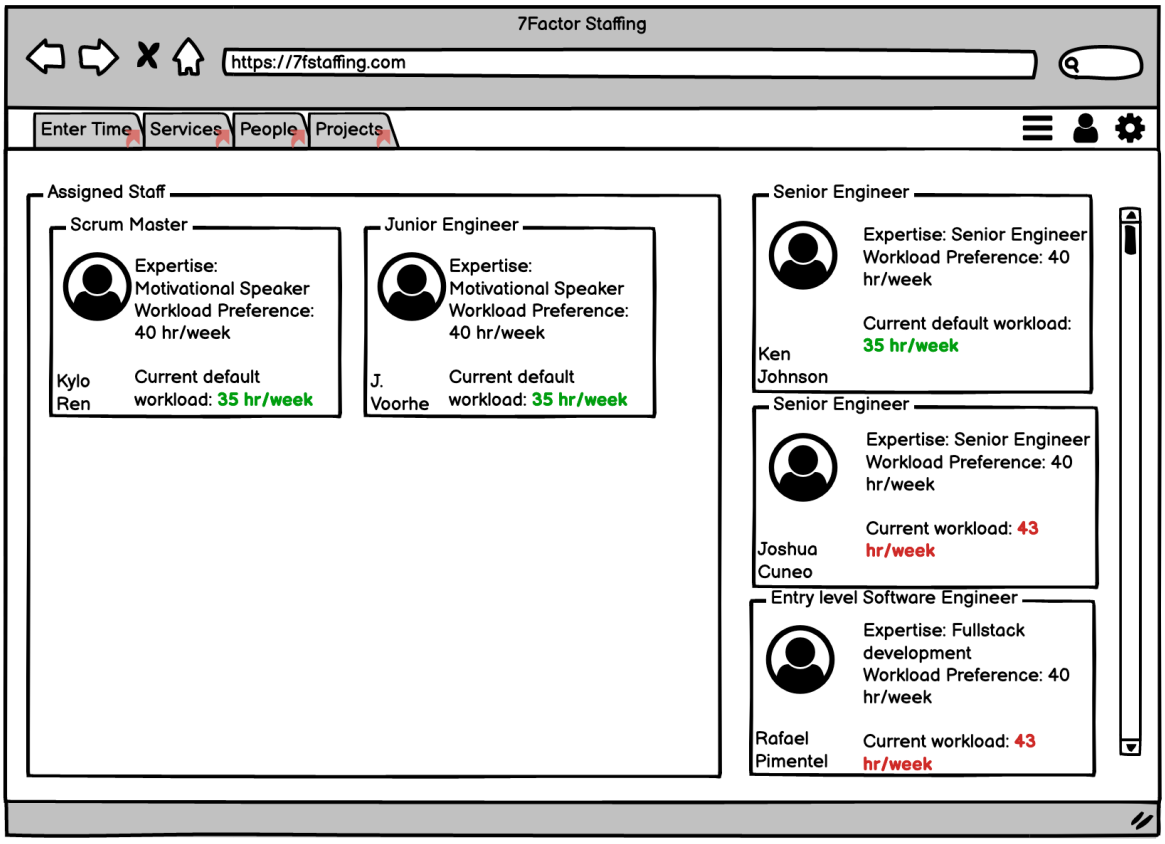

Figure 11: Wireframe of the screen to view all employees within the company within the project information screen

This extension of the Project Detail page allows the administrator to drag and drop available employees from the right side of the screen into the project staff on the left side of the screen, or likewise remove an employee, as shown in figure 11. In this version, each card represents an employee and shows useful information such as their current workload and expertise.

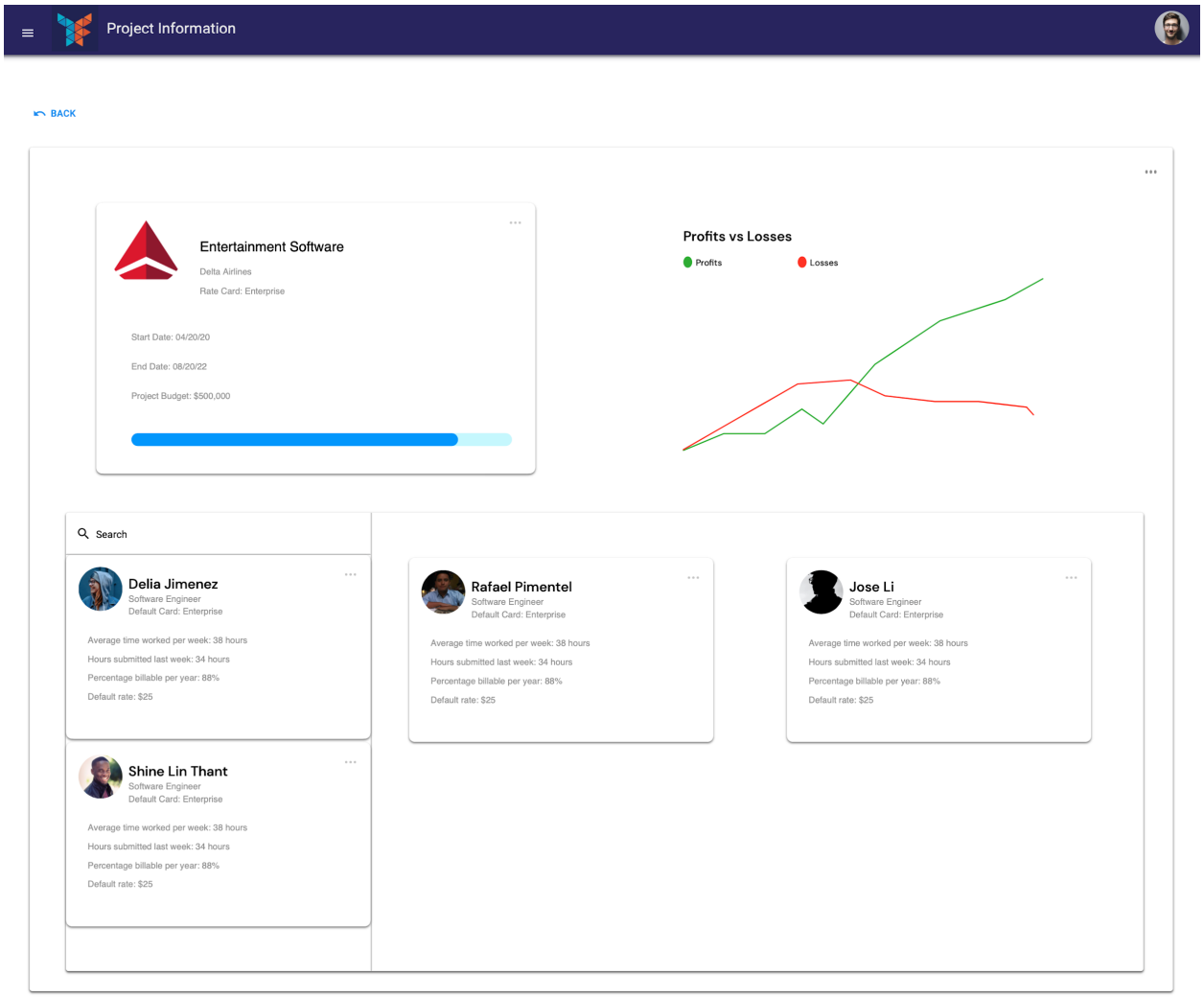

Figure 12: Updated mockup of project details screen

This screen also remained largely the same in the final mockup. The design is heavily dominated by cards as shown above.

Rate Cards (Services)

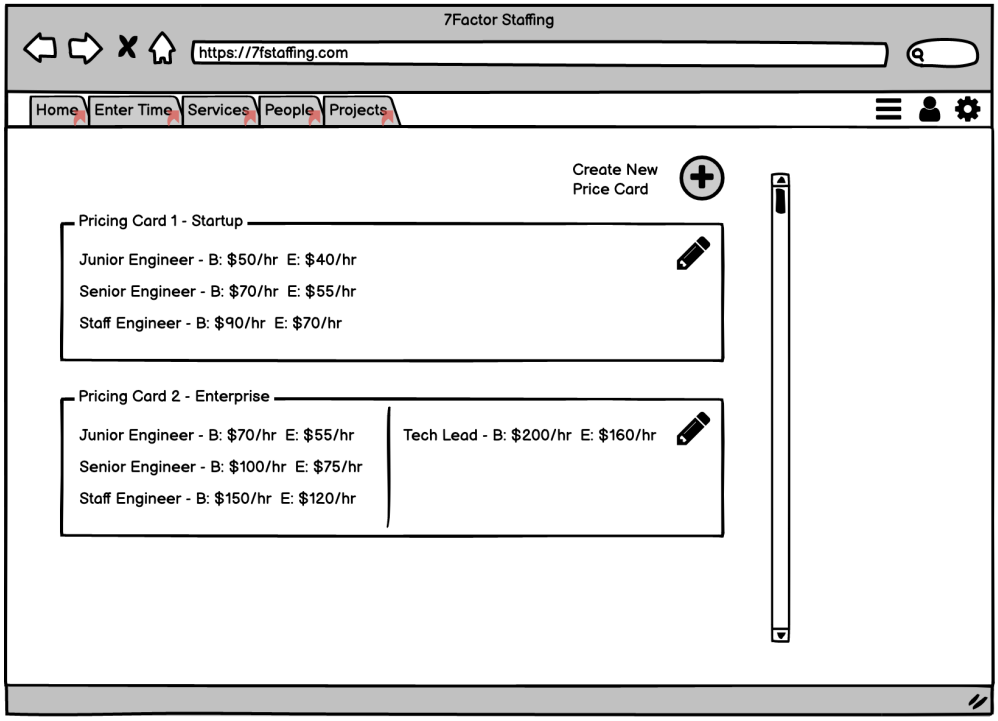

Figure 13: Wireframe of the Price Card creation with roles and rates

This screen allows the administrator to create new rate cards. A rate card contains a set of Positions, where a position specifies the job title, billable hourly rate of the consultant to the client and the rate paid out to the consultant. It facilitates creating a set of default rates for a project as specified in core requirement 2 and is also used to modify the existing billing rate of a consultant as specified in core requirement 3 and displayed in figure 13.

Task sequences:

1. Create a rate card: Log in → Go to Services page → click the plus icon → add as many price levels as needed → click submit
2. Edit a rate card: Log in → Go to Services page → click the edit button of a service → add or edit price levels as needed → select when the changes will apply → click submit

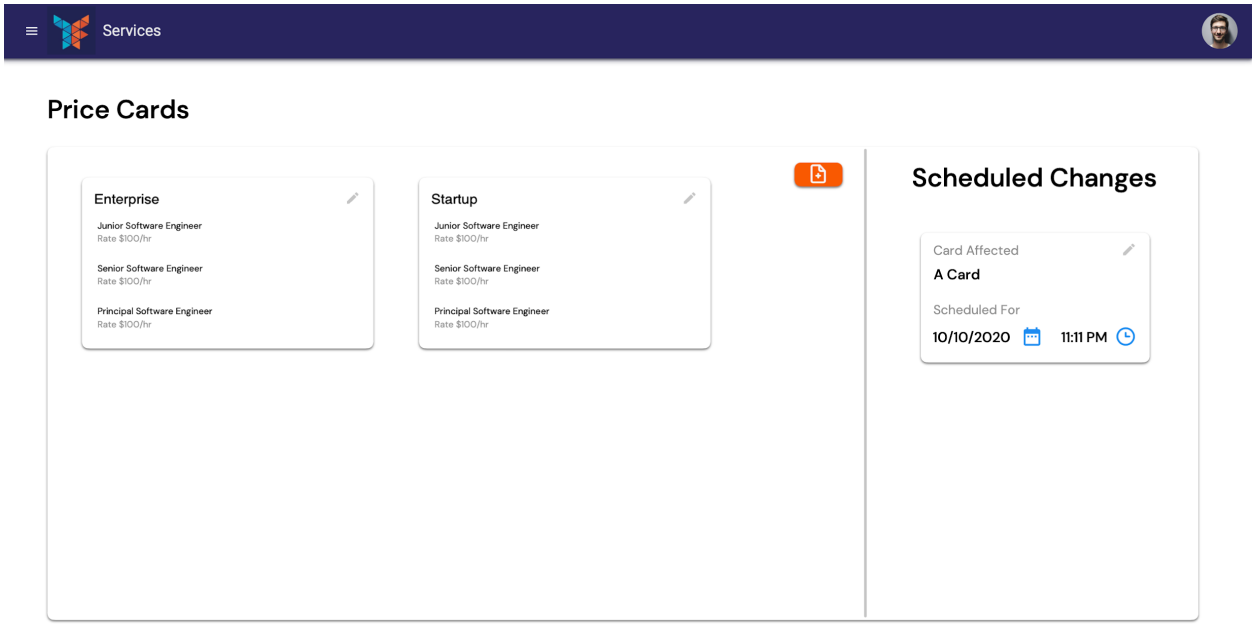

Figure 14: Updated mockup of the create a rate card screen

We changed the final design of the rate card (i.e. price card or service) page to be more consistent with the rest of the user interface by using cards inside frames as shown above. The left side shows all available price cards with the various positions embedded. On the right, the screen shows any changes scheduled to take effect on any of the rate cards in the future.

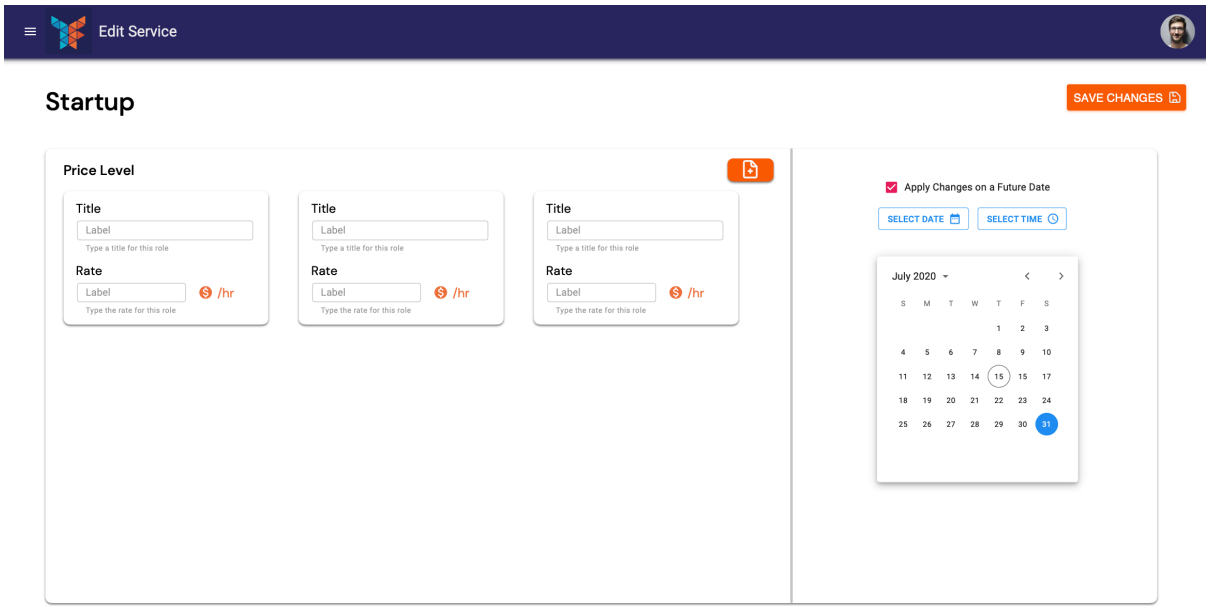

Figure 15: Updated mockup of the edit a rate card screen

We also designed a separate “Edit Service” screed to support editing rate cards, shown in figure 15. Most importantly, the right column lets the user schedule the change to take effect at a specific time in the future.

Stretch Goal - Home/Insights

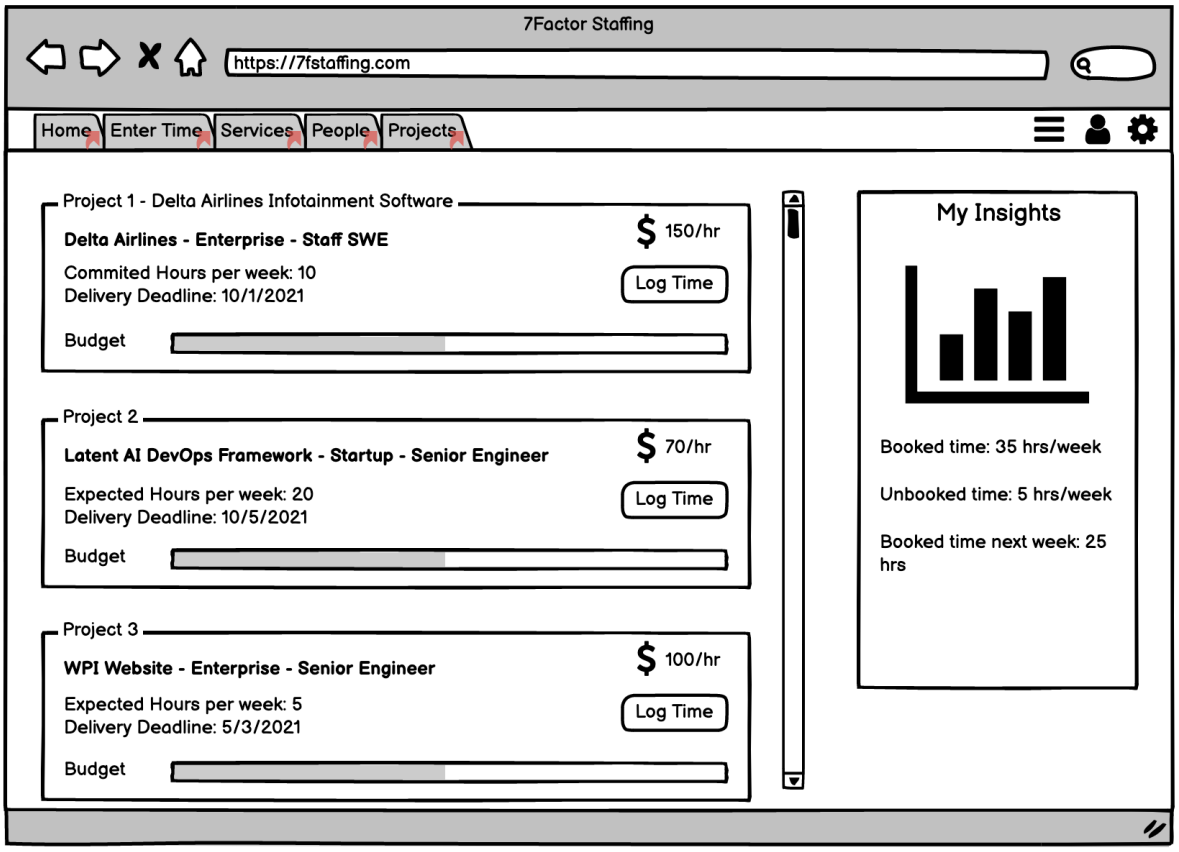

Figure 16: Wireframe of the insights screen for information of the company

This page shown in figure 16 is meant to give every user access to information about the projects they are assigned to and let them easily reach common tasks such as entering time. It would be the “default” screen after logging in. This page is a stretch goal and represents our idea of what the platform could look like in the long run. We did not create a detailed mockup because it would reach beyond the scope of this project.

Technical Requirements

Framework Requirements

There were three major factors that influenced our choice of framework. First, that it is relevant and up-to-date with no risk of going obsolete soon. Second, whether the 7Factor team is familiar

with it since we would hand off the codebase to them. Last, that at least one of our team members already has some experience working with it.

For the back-end, we chose the Spring Boot framework. It is a modern, popular framework that supports model-view-controller applications in the Java programming language (Ramacciotti, n.d.). Kenn, our tech lead, is a Java developer, and all of our team members are familiar with Java so Spring Boot was a fitting choice.

For the front-end, we chose the React framework. It is an industry standard and a very widely used and well-supported front-end framework, so it was a natural choice for our platform.

Architectural Concerns and Design Patterns/Conventions

To standardize the architecture of the application and increase the separation of concerns, we followed the model-view-controller (MVC) architecture. Specifically, we used Java objects along with Spring Boot annotations to define the database schema and entities, which comprised the model. The next layer was the data access layer, which included the Spring Boot repositories responsible for communicating with the database. Next up, the service layer encapsulated the business logic of the application and exposed methods that directly perform many of the final features of the application such as assigning an employee to a project. Last, the controller layer exposed the REST Application Programming Interface (API) that was responsible for communicating with the front-end of the application through the HTTP network protocol.

The MVC architecture guidelines suggest that each layer should be defined by an interface and separately implemented by a class. Early in the development of the application we had a discussion with 7Factor about whether to incorporate this element into our design. On the one hand, following the guideline would have achieved maximum separation of concerns and maintainability by ensuring that the interface of each layer stays constant while the implementing classes can change over time. For example, when an implementation changes, the underlying object can be swapped transparently to all other layers since the interface is what defines the functionality. However, the 7Factor team noted that this separation will incur significant development overhead for our application because each design change will have to be carried over to both the interface and implementation. Moreover, because of the relatively small scale of the project, the extra overhead will not provide significant benefits. For example, an implementation of a service-layer object can be swapped by simply transferring the Spring *@Service* annotation.

Team Organization

To organize, prioritize and track engineering tasks our team followed the SCRUM software development methodology. We hosted our code in the GitHub platform for version control. We used the online project management tool Clubhouse to create an online board on which we posted entries representing different tasks. Those entries were organized in different columns, representing the current stage of development they were in. The different columns were Unscheduled, Ready for Development, In Progress, Blocked, Ready for Acceptance, and

Completed. Each entry would move sequentially through the different stages as we made progress on it and would be tagged with the person or people working on it.

A few weeks after we started working on the project, it became apparent that using the online board alone was not enough to ensure we were making sufficient progress and were prioritizing the right features. So, we decided to hold daily SCRUM meetings and sprint planning meetings at the beginning of each week. During the daily meetings, we would simply state what we worked on, if any of our tasks were blocked by another task and what we planned to do next. These meetings lasted about ten minutes. They proved to be a very effective tool to not only keep us in sync with each other but also encourage us to make consistent daily progress.

On the other hand, the sprint planning meetings helped us understand what the overall status of our application was and prioritize the features that would make the most impact. At the beginning of the planning meetings, we started by putting down all different tasks we could come up with as entries in the Unscheduled column. Then, we discussed what we wanted to achieve by the end of the sprint and how each individual task would contribute to the overall goal. Last, we moved the tasks identified as most important to the Ready for Development column. Each sprint planning meeting lasted about one hour.

In addition to prioritizing and tracking, receiving feedback was crucial to ensure our code was of high quality and met the functional and qualitative specifications put forth by our sponsor. We obtained feedback in two different ways, namely pull requests and demo meetings. Demo meetings played an important role from the beginning of our project, when we presented the wireframes we designed for our user interface and the technologies we were planning to use. They were an opportunity to talk directly with our sponsors, listen to their feedback and see their reactions to our various ideas and deliverables.

Pull requests, on the other hand, provided a way to consistently receive written feedback on our code. In the beginning of our project pull requests occurred about as often as demo meetings and spanned across tens of files and commits. We had a difficult time breaking down the foundations of our application to individual working pieces, and thus only delivered large portions all at once. However, the large size worked against the spirit of gradual feedback that pull requests are meant for and made digesting the code difficult. As a result, we gradually reduced the size of pull requests to one per task, and published one almost daily. In that way, each pull request only spanned across a handful of files and was easier to review. With a combination of fewer demo meetings and very frequent pull requests we were able to get consistent feedback for the implementation and overall direction of our project.

Database and Back-End Design

One of the biggest design decisions made during our project was how to create the back-end. After we first met with our client and sponsor 7Factor, we knew what structure we needed by looking at our User Interface mockups. With the mockups on hand, we held a meeting to decide what type of data we needed for each page and how that would translate into SQL tables. We then translated this into an Entity Relationship Diagram:

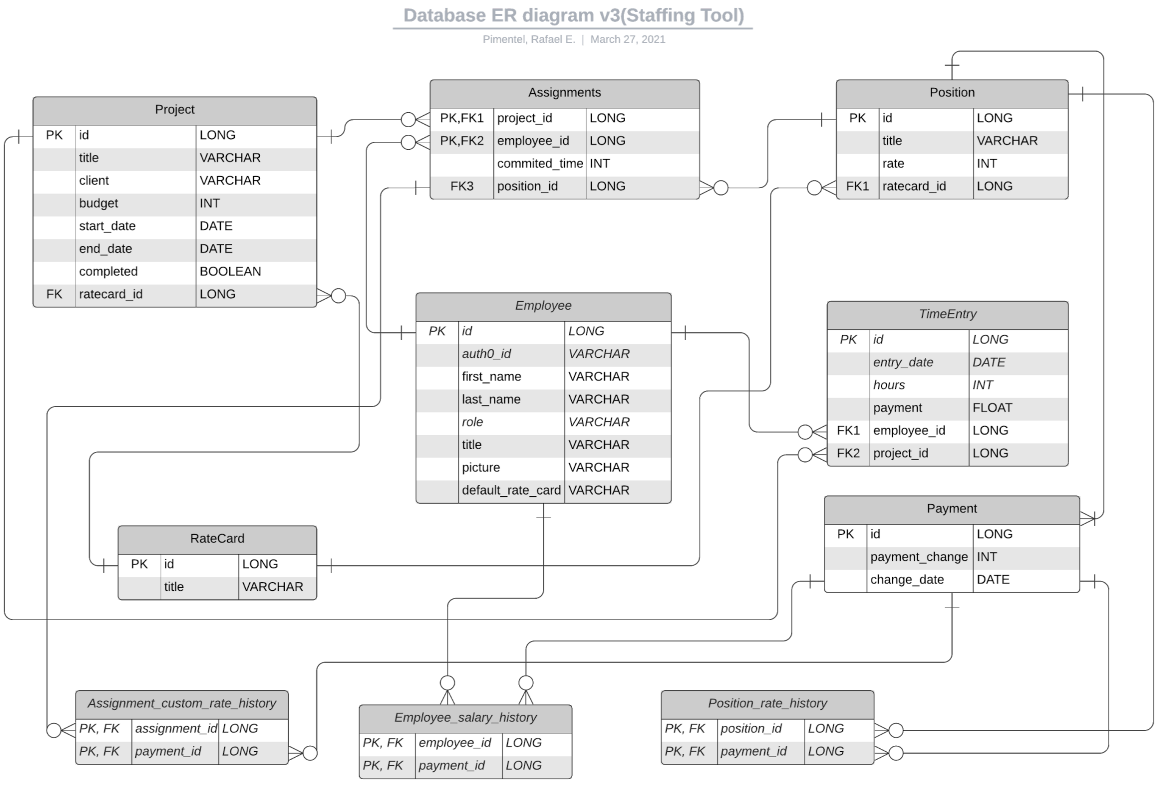

Figure 17: ER Diagram of our Schema

While we created the diagram, we made critical decisions on the relationships between our data based on the needs of our sponsor and made sure to gather enough resources to be able to replicate everything in Spring Boot using JPA, the Java Persistence API, which allowed us to define what objects need to be persisted in our application. In the next sections we will explain the rationale behind each table in our database and the relationships they had.

Price Cards, Positions and Payments

One of the principal things we needed to figure out was to create price cards. They are needed to assign a default set of hourly rates for a project. Each price card has a position, which is used to assign a payment rate for an employee assigned to a project. We created a one-to-many relationship in between a project and a price card, so many projects can point to the same price card in the database, but each project can only have a single price card. The other relationship this table has is a one-to-many relationship in between a position and a price card. This way a price card can have many positions, but a position can only point to a single price card. Additionally, positions have a one-to-many relationship with payments. A payment is a tuple of `(float amount, Date date effective)` and is used to represent a salary or hourly rate that may change over time. A position holds multiple payments, each payment with a unique date, and can use those dates to determine which payment was valid at a given point in time.

List of Payments and Support for Scheduled Changes

One of the required features for payments is the ability to update them at a certain date and time, and to be able to store a log of these edits. For this reason, each entity that has a payment holds a collection of Payments as opposed to a single instance. Every payment object has a Date attribute that determines when it becomes effective. So, when a parent entity needs to determine which payment is valid at a specific date and time, it iterates through the collection of payments and finds the one with a date that is closest to but less than the required one. This approach allows us to schedule a future change simply by adding another payment to the existing payments list without having to worry about checking for and managing events in the future. We do not need to implement any interrupts or polling-based services to continuously monitor and update a central “current payment” object. The biggest drawback of this approach is that every `find` call becomes a search, which is more computationally expensive and less scalable than a simple `findById` call. However, by sorting the payments list by date we can minimize the query execution time to the logarithm of the number of elements in the list, which would be sufficiently quick for most deployments.

Projects, Assignments and Employees

Projects and their relationship with employees are the core feature of our back-end. The assignment object creates an association between projects and employees. This way employees can bill hours for a specific project. Each assignment contains basic information about the employee in relation to a specific project, such as a custom payment rate, the assigned time per week to the project and the position of the employee in the project. To do this, we created a many-to-many relationship by adding the assignments table. An assignment points to both an employee and a project. This way projects and employees can have many assignments, but each assignment can only point to one project and one employee at a time (“Many-To-Many relationship in JPA”, 2021).

Time Entry, Projects and Employees

A time entry is the billing of the hours worked by a specific employee for a specific project. Each time entry contains information such as the date of the time entry, the number of hours worked and the final payment amount. We created one-to-many relationships for projects to time entries and from employees to time entries; this way we can link a time entry to a specific employee and project. This way a project can have many time entries for one or more employees and an employee can have many time entries for one or more projects.

Unit and Integration Testing for the Back End

To ensure the back-end of our application was high quality we had to address three major concerns. The first concern was that the database design was functional and accessible. The second concern was that the service logic worked as expected. Last, the third concern was that the controllers were capable of accepting all the required parameters and could correctly

interface with the rest of the stack. We used a combination of unit tests, integration tests with mocked HTTP requests and mocked database repositories, and integration tests with functioning databases to ensure all the components performed correctly.

The simplest form of tests we used were unit tests. We mostly used them to test service logic and consisted of calling service methods directly and writing mocked repository calls to simulate the database access layer. In that way, we could isolate the methods we were interested in by explicitly defining the response of the database access layer without the complexity of mocked HTTP calls. We used unit tests for methods with higher complexity than most.

With the extra complexity of mocked HTTP requests, our controller-focused integration tests tested the controllers and services but not the database repositories. They helped us verify that we could pass in parameters and objects through HTTP requests and receive the correct responses. In this level of testing we used mocking for the repositories to pre-define the database responses for each test (“Testing the Web Layer”, n.d.). Those tests were especially useful when attempting to pass in complex objects through HTTP and when expecting various HTTP error responses.

Our most sophisticated integration tests used a real database to test the entire application from the controller all the way down to the database. We used a framework called TestContainers (“Testcontainers”, n.d.) to instantiate a real database container on the local machine at the time of testing specifically to be used by the tests themselves. The testing class defined setup and teardown methods that were called before each individual test to ensure each one had an identically populated database to interact with. The database container was destroyed after all tests finished running. These tests did not use mocked HTTP calls to avoid the added complexity. This layer proved immensely useful because it allowed us to test all the way down to our database schema. Also, the setup method provided a good example of how the back-end can be used to populate the database in the correct order which was very helpful for the front-end team. Last, these tests were excellent at revealing integration issues throughout the application and became the go-to quick testing suite before merging a pull request; in other words it served as our “smoke test”.

Last, we used test coverage reporting to measure the reach of our testing. Coverage reports helped us reveal branches of the application we had not tested at all and direct resources towards them. However, we did not rely on this tool because high code coverage alone does not mean that the application is well-tested and will always behave as expected.

Auth0 Implementation

One of the first features brought up by 7Factor, was user authentication using Auth0. Auth0 is an identity management platform for web, mobile and internal applications. Determining what was the best way to implement it in our project was challenging, since we have to take into account multiple variables, such as protecting our React application, protecting our Spring Boot API and creating a connection in between the employees in our database and the Auth0 users.

Our first task was deciding between authentication and authorization in each layer of our project. After doing some research we decided that our first step was implementing authorization in our Spring Boot API.

Securing our REST API

Once an Auth0 account and a corresponding tenant was created for us by 7Factor, we implemented a new API configuration in the Auth0 dashboard, setting a name and identifier. Auth0 suggested using the RS256 signing algorithm to sign JSON web tokens. Every request to the API is accompanied by a unique signed token. The token is a string of characters that is created in such a way that the API can validate it and ensure the requestor is not trying to impersonate other users ("JWT signing algorithms", 2020). This identifier will help us make secure requests to the API from other endpoints inside our Auth0 tenant. After adding information about Auth0 inside our *application.properties* Spring Boot configuration file, we started creating our security configuration using the Spring Security framework. By creating a class that extends `WebSecurityConfigurerAdapter`, which is a class that gives us a default security configuration for our project, we were able to customize it to protect specific endpoints of our API. This configuration ensures that every valid request to our API contains a JSON web token. Additionally, we created a decoder that allows us to check that the JSON web token is correct, and comes from the right issuer and audience. Finally, we enabled cross origin resource sharing, which allows our application to load resources from sources outside the API itself, such as making HTTP calls from our React front-end ("Spring boot Authorization Tutorial: Secure an api (java)", 2021).

Authenticating a React Application

Next, we created a new configuration for a single page application in Auth0. After assigning the application a name and a domain, we configured a callback URL so Auth0 knows where to redirect the user after successful authentication, and a logout URL so the application knows where to return to after a successful logout. Additionally, we configured the "allowed web origins" field to be able to refresh the authentication token provided by Auth0 ("Auth0 React sdk quickstarts: Login", 2021).

In React, we first added the Auth0 Software Development Kit (SDK). Then, we wrapped our React application in the provided Auth0Provider component, which requires us to pass as props, the domain, the client ID and the redirect URL to correctly transfer the user after authentication. The Auth0 SDK provides methods that can be called from either React hooks or class components. These methods allow us to perform various actions, such as logging in and out, getting user information and tokens from the Auth0 management API, or getting information from any custom API defined in our Auth0 configuration. After this step, we can request a token from Auth0 so we can add it in our HTTP requests configuration and get authorized by our API ("Getting started", 2021).

Creating an Association Between Employees in our API and Auth0 Users

Auth0 uniquely identifies users with an auto-generated id which is internally named `user_id`. Similarly, our database identifies users with a field inside the employee entity called `id`, which we will refer to as the native ID. The two ID fields achieve the same purpose but are generated and managed by two different systems. In order to relate users from Auth0 to users in our database, we saved the `user_id` inside our employee entity in a field called `Auth0_id`. After a successful login attempt, Auth0 returns the `user_id` to the front-end. The front-end can then use the `user_id` (i.e. `Auth0_id`) to request the native ID of the user from the back-end and then store it in React for making API calls. Calls such as saving new time entries or getting projects for a specific user required passing in the native ID.

Challenges Faced During the Project

During development of the front-end, we encountered issues when creating entity objects, such as a project or employee, and utilizing the API to connect to the database. First, we ran into issues when dynamically rendering elements based on certain information that is present within the database. Dynamic rendering is the process of using information obtained through API calls to pass information down from a parent component to a child component so that this child can display the correct information. We used dynamic rendering for many components in the front-end such as employees, projects or rate cards. We had difficulty finding ways to dynamically render out components for a given screen.

Another issue was that we were initially working with function-based components and hooks which do not allow for states in react but instead utilize hooks for maintaining states. Function-based components are stored within a function rather than a class and have access to new functions from React 16.8 such as `useState()` and `useEffect()` (“React – a JavaScript library for building user interfaces”, n.d.). These new functions are intended to make code cleaner and replace some outdated features of class-based components such as the `componentWillMount()` (“State and lifecycle”, n.d.) lifecycle function. The other important difference between the class-based components and function-based components is that we lose access to lifecycle methods with function-based components. Lifecycle components are important because they trigger API calls to be made. When a component renders on the screen, it will also call its corresponding lifecycle function. In our case, before a component renders, it will make an API call and populate on-screen fields based on the API response.

More specifically, we had problems making sure that we had the proper information to render components with the API call. We had to utilize the react lifecycle function `componentWillMount()` (“State and lifecycle”, n.d.) to ensure that the component we want to display will render with no errors. This function is called before the component renders and triggers the API call. We then use the JSON object from the API response to set the states of parameters within the parent component and utilize the map function. We use the map function to map out a child component for each value stored within that state from the response JSON. For example, we can set the value of a state equal to an array that contains a JSON object of

each returned value from the API call. With this storage array we can use each value to populate information for that dynamically rendered component such as an ID or Title.

Another difficulty that we had with the front-end was getting the API calls to work seamlessly. When making calls, we had to obtain a JavaScript promise from the API call and then use a built in function called `then()` to get the value from that promise. This process becomes challenging in combination with the need for obtaining an Auth0 token for authorization and dynamic rendering as discussed above. In addition, forming valid API calls in the front-end was a challenge since we had to place information obtained from user forms into objects that the back-end could recognize. Consider the following example: an employee has a child rate card object which we want to send back in its parent object. To complete this operation, we utilized another built in function called `JSON.stringify()` which would convert the object to a string when we were saving the data, and then when we would create the object POST we would use `JSON.parse()` to parse the string and recreate the object (“Getting started”, 2021).

Last, including 7Factor staff in our sprint planning meetings would have allowed us to better prioritize our tasks and receive feedback, help and technical mentorship more consistently. Initially we did not believe this arrangement was possible due to time zone differences with our technical lead, Kenn, but we later found out that it was possible.

Results

Technical Stack and Technologies Used

The technologies we used for the development of our project were React with Material-UI (“UI: A popular React UI framework”, n.d.) for the front-end. Material-UI ("UI: A popular React UI framework”, n.d.) Is a library that provides premade react components that look nicer than the base components within React and possess prebuilt functionality. We also used Spring Boot for the API and Postgres for the database. We chose these technologies because at least one of our team members was familiar with them and our technical lead Kenn had prior experience using them. We also found them to be overall best suited to the needs of our project.

User Interface

The final user interface is presented below. We used our detailed mockups as a basis and recreated them in React alongside Material-UI (“UI: A popular React UI framework”, n.d.) it is important to note that several screens do not possess final styling because we prioritized the functionality over the appearance.

Figure 18 shows the login screen. After implementing Auth0, we chose to maintain a more minimalistic look than the original mockup. On this screen only the 7Factor logo and a login button are visible and would direct the user to the Auth0 login page.

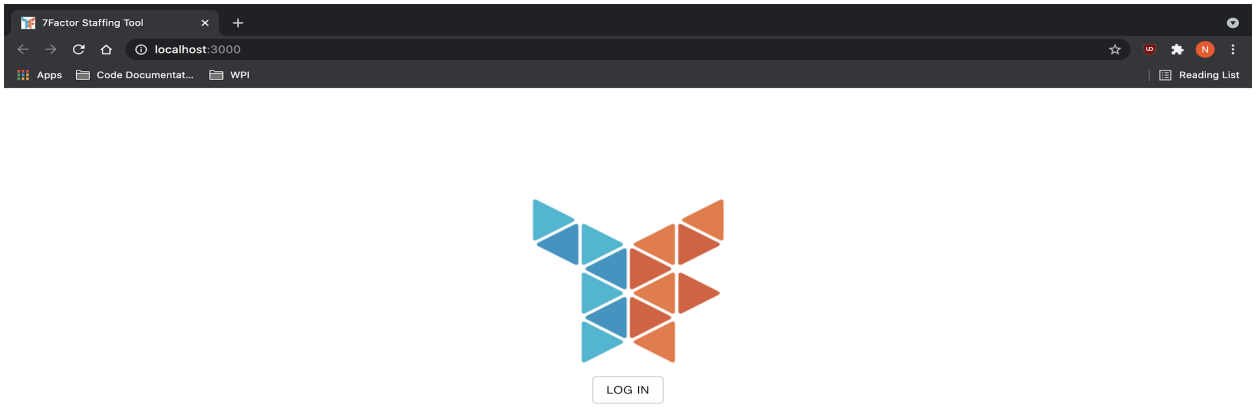

Figure 18: The final login screen of our working application

Upon logging in, the user is taken directly to the time entry page which is presented in figure 19. Each row in the view corresponds to a project the particular employee is assigned to. By default the employee would be presented with a row for each project associated with them.

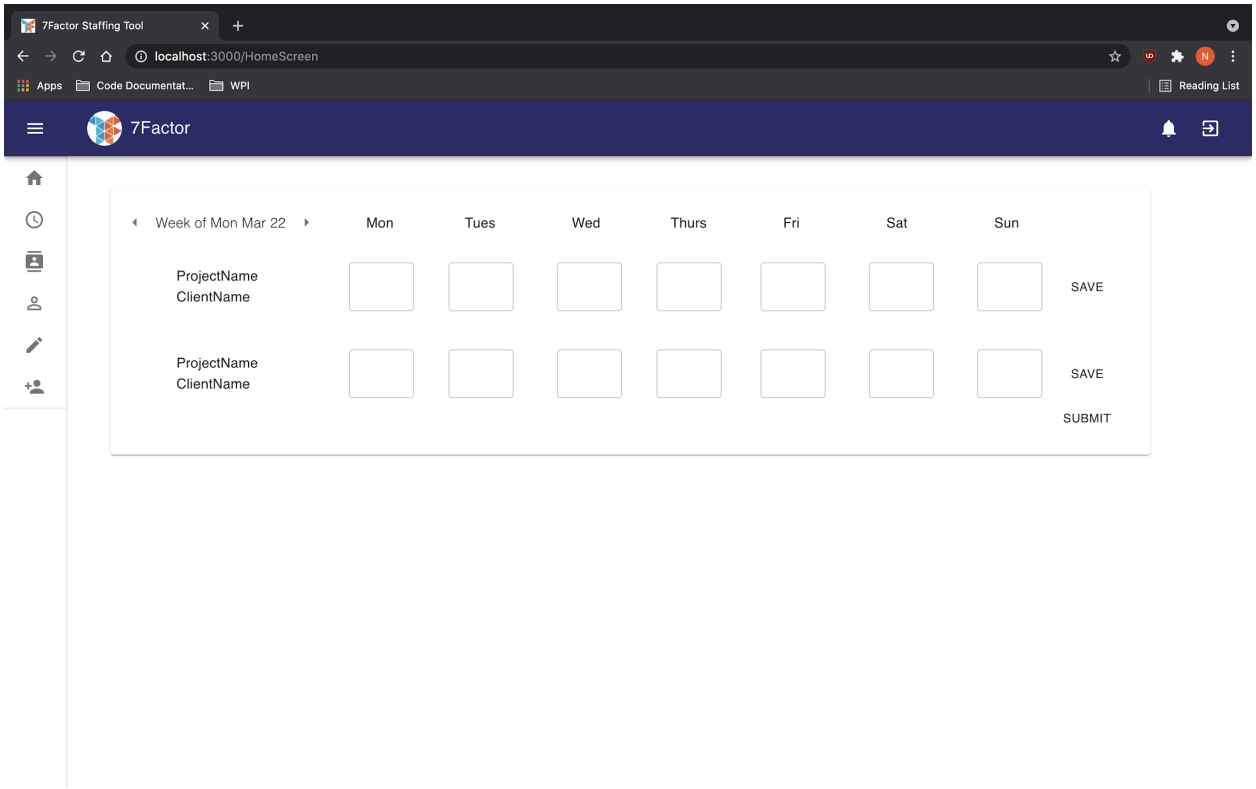

Figure 19: The final time entry screen

The next screen, shown in figure 20, is the project creation screen. In a production system it would be only visible to users with administrator privileges. However, we did not have time to implement role-based authentication. One key note is that we did not implement the functionality of assigning employees to projects on this screen as we originally planned. Instead, we separated it out to a new screen.

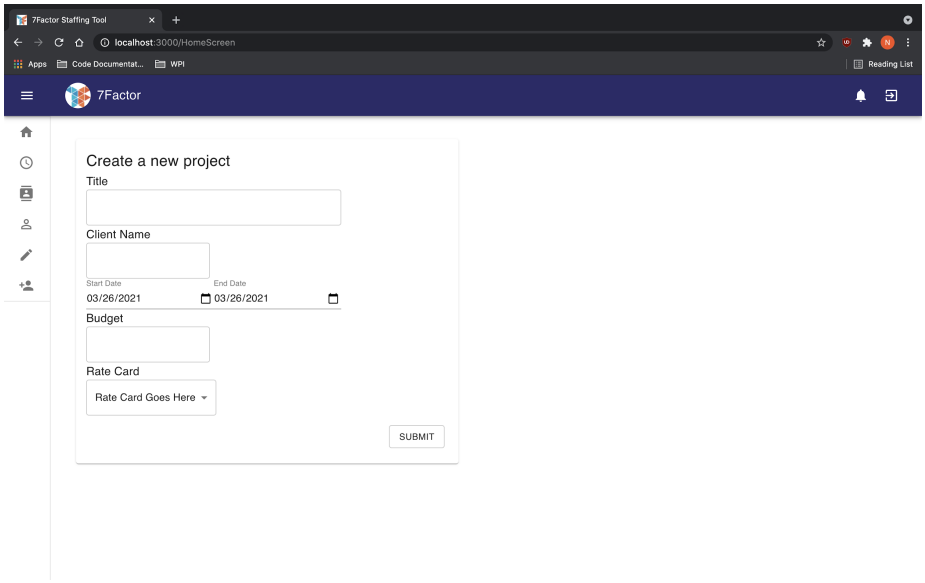

Figure 20: The final “create a new project” screen

Following project creation, we also have the employees screen. This screen gives the ability to see all the employees rendered in the database as well as assign an employee to a given project as seen in figure 21.

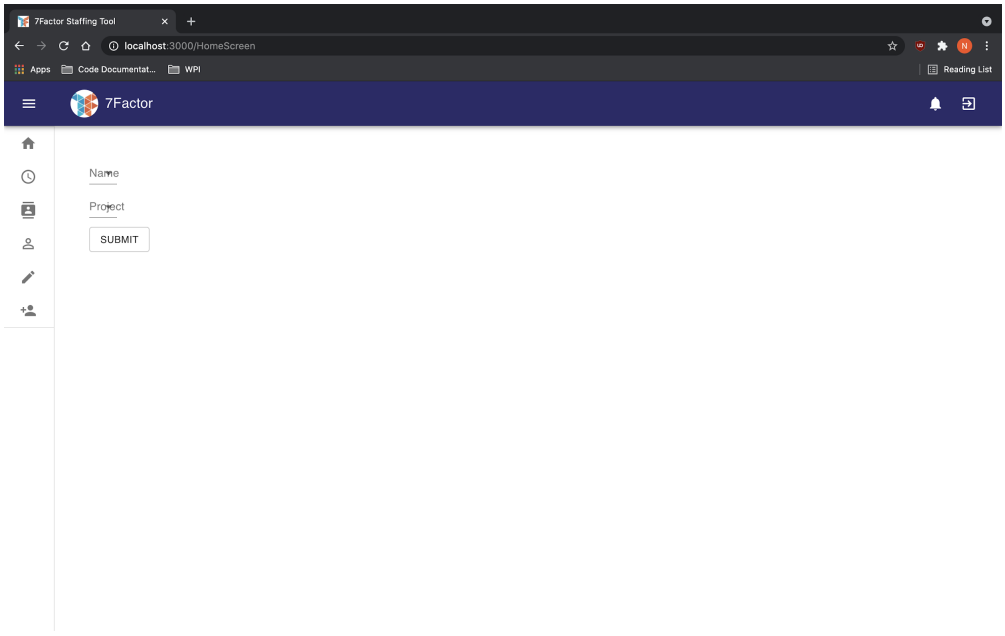

Figure 21:The View all employees screen, when no employees exist in the database

Next, we have the “add a rate card” screen. This screen allows for a new rate card to be created with the correct information. It dynamically renders rows for each role that this card possesses.

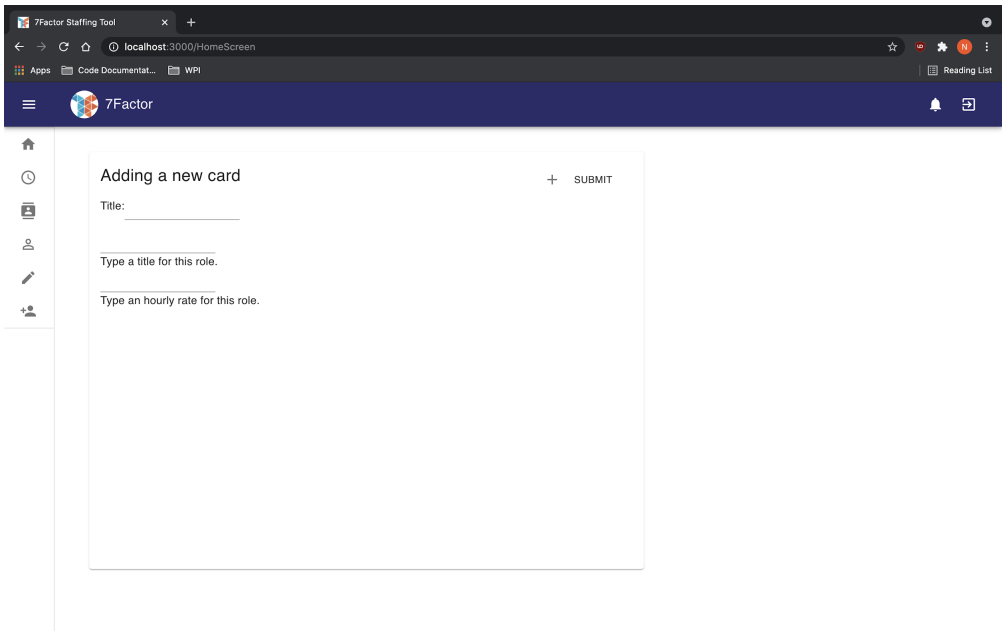

Figure 22: The final UI of adding a rate card

The next screen, shown in figure 21, allows the administrator to create a new employee with some basic parameters. These parameters include their first and last name, their job title and their default rate card.

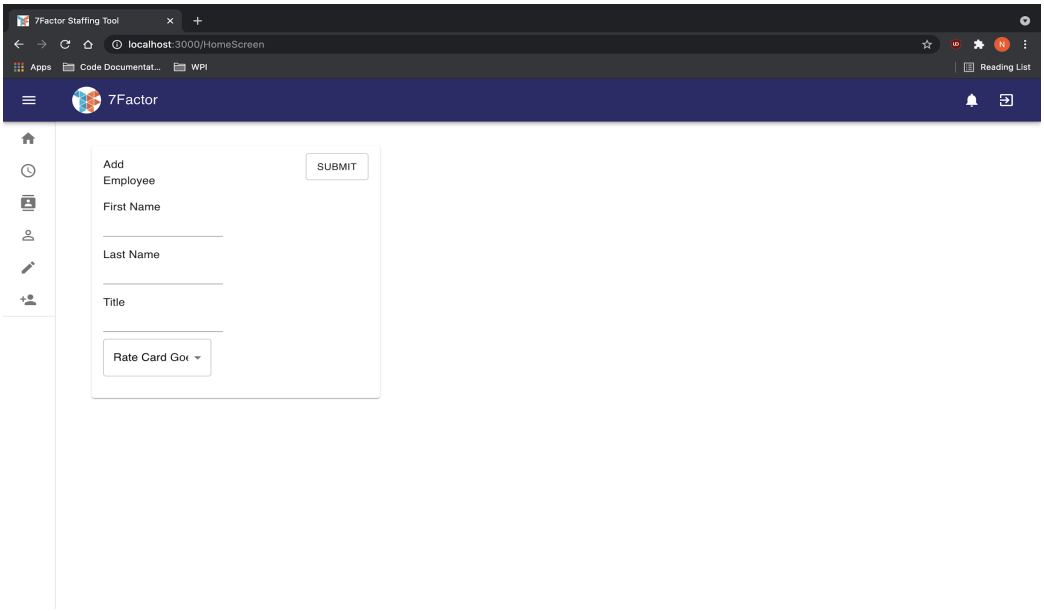

Figure 23: The final screen for adding an employee to the database

User Interface Summary

We wanted to maintain the designs that we had created in the mockups as well as the renders that we made in Sketch for our final user interface. We utilized a minimalistic design and paired it with the 7Factor color scheme.

User Testing Results

We conducted user testing to understand the effectiveness of our User Interface (UI). During each test, we asked the test subject to complete specific tasks and rate the ease of use of the interface. We did not provide directions on how to perform the tasks and instead observed if the subject could intuitively navigate the UI. The specific script that we used is available in Appendix B. We conducted the tests over a zoom call where the only participants were the facilitator, two observers and the test subject.

Our Roles

Panagiotis: As the facilitator, he conducted the test by reading the script and interacting with the test subject directly.

Nicholas and Rafael: As observers, they took notes while Panagiotis conducted the test.

Test Observations and Participant Feedback

User 1

The user generally expressed positive comments for the interface. They liked the icons and minimal aesthetic. They also noted that some UI elements were inconsistent across screens and could be more uniform throughout the entire application.

Creating a new rate card:

When creating a rate card, the user did not initially understand that the “+” icon was for adding more rows to the card. They also thought that it was unclear what the difference between the title of a rate card and a title of a role was within the rate card. They suggested moving the “+” button to be under each row and making each role an individual card.

Creating a new employee:

When creating an employee the user had found it simple to create a new user. The task was easy to complete but the user had comments on delinking employees and a given rate card.

Creating a new project:

The user was able to create a project with little to no difficulty. Their only comment was rather than keeping the “Client” field as a free text input, perhaps change it to a dropdown menu that would list all of the pre-registered clients. This change also implies creating a separate entity set to represent clients and a new UI flow for creating and editing them.

Entering time:

Entering time was overall simple for the user. However, they disliked having individual save buttons for each row. They suggested having a single button to save all rows. In addition, they suggested adding fields to display the total hours worked each day and in total for the entire week. Furthermore, they would prefer to only see the projects they were active on during that week rather than all projects they are assigned to. They also suggested having two additional time entry rows for registering time spent on unpaid projects (i.e. the “bench”) or on paid time off.

User 2

In general user 2 had no issues completing the four tasks. However, they pointed out that some button labels were missing and others did not correspond to the functionality of the button. They also suggested improving the user experience by modifying how days were labeled in the time entry screen and adding more tooltips and labels.

Creating a rate card:

User 2 did not understand that one rate card can have multiple roles on it, instead thinking that a rate card was a single role. The facilitator stepped in to resolve the misunderstanding. The user did not know what the plus button did on the screen. They suggested adding tooltips and labels to clarify what the different entry fields and buttons do and make the task flow easier.

Creating a new employee:

The user completed the task easily and without facing any challenges.

Creating a new project:

The user completed this task without issues, but suggested styling changes to make the user experience better and improve the look of the page.

Entering time:

The user completed the task of entering time successfully. However, they had comments similar to those of user 1. They would have liked to see the total hours worked in a given week, day or project displayed alongside their time entries. In addition they wanted to see the exact dates displayed under each day label. For example, they would like to see “1/1/2020” displayed under the word “Monday”.

Overall Assessment and Suggestions

One of the main issues that users noted throughout our tests was the lack of clarity about what different icons or buttons represented. To address this challenge one user suggested adding labels and tooltips. These additions would provide more information about the corresponding UI element and improve the intuitiveness of the interface. Also, both test users wanted the total amount of hours worked for each day, project, or the week overall to be displayed alongside their time entries. Last, the users mentioned that the components and styling could be more consistent throughout the entire product.

Overall the user testing was very successful. It provided us a glimpse on how users would interact with our interface when attempting to complete a task and how they would go about completing that task. In addition, we were able to receive constructive feedback about the flaws of our design and make significant usability improvements.

Recommendations and Future Work

There were several features that we originally planned for but were not able complete within the time frame of this project. We completed most of the core functionality such as creating employees, project and rate cards, assigning employees to projects and submitting time entries. We spent much of our time implementing the database and back-end which support a significantly larger portion of the functionality than the front-end. For example, the front-end does not support editing most entities after they have been created. We have several suggestions on what could be added in the future, regarding both the front-end and back-end.

The first additions to the front-end could be basic editing of rate cards, employees and projects. All of these features are implemented in the back-end but currently not in the front-end. More advanced object validation, error handling and user feedback when errors occur would significantly improve the reliability and user error tolerance of the application, in addition to providing a better experience for the user.

One feature that was mentioned during the demonstrations and user testing sessions was the ability to hide some of the projects in the time entry screen, based on which ones the employee is actively working on. This would improve the user experience since there would not be as many rows visible on the time entry screen. In addition, the ability to set a default template for new time entries would improve the ease of use since many time entries follow the same pattern. For example, this default template would autofill the time spent on a project. Thus the only step the user would make is to login and approve the time they spent working this week instead of filling the entire row.

It would be beneficial to perform a styling overhaul in the front-end and implement dynamic window resizing. Currently, the UI does not support smaller sized windows and mobile devices. Implementing resizing is important to improve the compatibility with different hardware configurations. It could be implemented by using percentages rather than raw pixel values when specifying margins. For example, `marginLeft: 10px` could be changed to `marginLeft: 10%`, which would maintain styling even if the window was resized.

Another important addition would be to modify the database, back-end and front-end to support a `Clients` table. In our implementation, a client is represented only as a free text field in each project entity. As a result, it is very difficult to track which clients have more than one active project. Creating a separate database table to store clients, along with the corresponding management interface, would pave the way for computing crucial aggregate data such as the total billable amount for a client and date range. In addition, we could store other important client information such as their address, e-mail and tax identification number.

To support scheduled changes in salary or rate information, our database currently holds lists of payment entities. The payments are associated with their parent object through a separate table that holds the ID of the parent object and the ID of the payment. This internal representation generally behaves as expected but requires an additional table for each entity that holds a payment list. For example, in our implementation there are three separate tables which

represent relationship sets of assignments, employees and positions with payments. Ideally, we want to avoid creating unnecessary tables and represent relationships with payments in a single table regardless of the type of the parent object. One way to address this issue is to create an intermediate entity that encapsulates the payment list. For example, we could create an entity called billing. The billing entity could have a one to one relationship to any other entity that needs to reference a list of payments, such as a position or an assignment. Then in the payments table we could store the billing ID in each row so the payments would be linked to a specific billing. This would create a one to many relationship between billing and payments, so a payment can reference one billing but a billing can reference many payments. With this structure in place, payment entities can only have billing entities as their parent thus only requiring one table to store the relationship set.

One of the technical issues we faced was loading lazily-fetched objects from the front-end. Consider the following example: the user wants to add a new position to an existing rate card. In the current implementation, the front-end would fetch the entire rate card from the back-end, modify the built-in position list and then post the entire rate card object (including the built-in position object) back to the back-end using a controller method. However, the built-in positions list is lazily-fetched from the database. That means that the service method has to explicitly fetch it before sending the object to the front-end. If not done carefully, this approach can defeat the purpose of lazy-loading. A more thorough approach would be to construct purpose-built Data Transfer Objects (DTOs) that only include the information that is strictly necessary to facilitate different controller calls. DTOs would also drastically increase the security of the application, protect objects against accidental changes and reduce the need for validating every object field at the controller level.

Implementing search in various parts of the interface, such as when viewing all employees or projects would improve the scalability of the application for larger deployments. For example, it would be impossible to scroll through a list of employees in organizations with over 1000 staff members to find the desired one.

The process of adding a new employee to the platform could be improved through Auth0. When the administrator wants to add a new employee, they can simply register the email of the employee so the system can send them an automated message. Through that message, the employee can set up their own profile. Once they are signed up, the application can look up the employee by the registered email in the database and save the Auth0 `user_id` for future lookups. Additionally, the implementation of the relationship between Auth0 users and our back-end employees needs to be finished so we can get information for a specific logged in user.

Last, the project needs to be integrated with the Continuous Integration (CI) pipeline of 7Factor. This step did not get completed because of time limitations that 7Factor team members had. Integrating with the CI pipeline, powered by the Concourse CI framework, would have sped up the development feedback cycle by ensuring every pull request passes quality standards and does not have any failing tests. It would also save significant time for both authors and

reviewers by running tests in automation instead of manually on the local machine of each contributor.

Notes for the Next Team

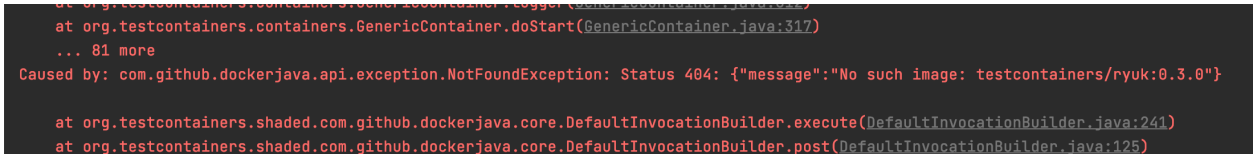

Figure 24: “No such image” exception when running `DBIntegrationTests` in a fresh environment.

When running the DB Integration tests on a new machine, an error occurs because docker cannot automatically pull the container “testcontainers/ryuk:0.3.0”. Manually running “docker pull testcontainers/ryuk:0.3.0” in the terminal fixes the issue.

After securing the API with authorization, we had issues accessing the Swagger-UI because the path is forbidden. This issue can be easily fixed by freeing the specific path from authorization for any GET requests.

We recommend reading the articles from Auth0 on how to authorize an API and how to use authentication in a Single Page Application. There are many discussions on how to identify users in an API using the `Auth0_id` (i.e. `user_id`) field that is accessible in the Auth0 framework.

Conclusion

7Factor requested a staff management application that would allow them to replace their current solution, TimeIQ. The new platform was required to support assigning consultants to projects, logging time and summarizing key project and business metrics. Through this project, we designed and implemented a new SaaS solution that supports the basic functionality for the core requirements of 7Factor and defines a structure that can be extended to support all of their needs.

Within the timeframe of our project, we designed and implemented the database, completed the majority of the back-end, created complete User Interface mock-ups and wrote a basic front-end application in React. We also implemented third-party authentication and authorization through the Auth0 provider. We spent much of our time implementing the database and back-end which support a significantly larger portion of the functionality than the front-end.

Bibliography

1. Arias, D. (2021). Auth0 React sdk quickstarts: Login. Retrieved April 02, 2021, from https://auth0.com/docs/quickstart/spa/react

2. Axios. (2021). Getting started. Retrieved April 02, 2021, from https://axios-http.com/docs/intro/

3. Chim, N. (2020, September 24). JWT signing algorithms. Retrieved April 02, 2021, from https://www.loginradius.com/blog/async/jwt-signing-algorithms/

4. Fejér, A. (2021, February 20). Many-To-Many relationship in JPA. Retrieved April 02, 2021, from https://www.baeldung.com/jpa-many-to-many

5. Ramacciotti, C. (n.d.). Spring Basics. Retrieved September 5, 2020, from https://teamtreehouse.com/library/spring-basics

6. React – a JavaScript library for building user interfaces. (n.d.). Retrieved April 03, 2021, from https://reactjs.org/

7. Slamic, T. (2021, January 20). Spring boot Authorization Tutorial: Secure an api (java). Retrieved April 02, 2021, from https://auth0.com/blog/spring-boot-authorization-tutorial-secure-an-api-java/

8. State and lifecycle. (n.d.). Retrieved April 03, 2021, from https://reactjs.org/docs/state-and-lifecycle.html

9. Testing the web layer. (n.d.). Retrieved November 03, 2021, from https://spring.io/guides/gs/testing-web/

10. Testcontainers. (n.d.). Retrieved November 02, 2021, from https://www.testcontainers.org/

11. UI: A popular React UI framework. (n.d.). Retrieved April 03, 2021, from https://material-ui.com/

Appendix

A. Recording Permission Form

You may have noticed the Zoom window. With your permission, we're going to record what happens on the screen and our conversation using Zoom. The recording will only be used to help us figure out how to improve the interface prototype, and it will not be seen by anyone except the people working on this project. This helps us too, because we do not have to take as many notes. There are a few people who are working on this project observing this session over Zoom. (They cannot see us, just the screen.) If you would, I'm going to ask you to sign a simple permission form for us. It just says that we have your permission to record you, and that the recording will only be seen by the people working on the project.

As part of our aim in providing a high quality application, we would like to make a recording of this testing session. Review of recordings will potentially improve our ability to make subsequent improvements to our prototype. Please read the following statement and, if you are in agreement, sign where indicated.

I consent to video/audio recordings being made of these sessions and to these recordings being used to aid the work. I understand that the conductors of this test will edit out from these recordings, or from descriptions of the recordings, as much identifying information as is possible.

Dated.....

Signed.....

Put a check mark next to the types of recordings that you consent to being made:

- ❏ Recording Video of you and the screen

- ❏ Recording Audio of you and Video of the screen

B. Test Script

Part 1. Introduction and Consent

Hi [test user], thank you for taking part in our study. My name is [host name] and I am going to be administering this test with you today. My teammates [teammate names] are also in the zoom call as observers. Just so I get everything I need to say I'm going to read from a script.

Today you will be helping us improve the interface of our web application for our MQP. We also want you to keep in mind that our design is still in development, and is not fully functional yet. This is not a test of you. You cannot do anything wrong here. Anything that seems like a mistake will just help us to improve our design.

As you go through the test, please talk out loud and share your thoughts as much as possible. It will provide us with valuable data to help us improve our website. Please do not hesitate if you think you may hurt our feelings. That is encouraged, any insults now will lead to improvements later.

If you have questions at any time, please ask them. We may not answer right away since we want to see if users can learn and understand our site. But if you have any unanswered questions after the test we will be happy to respond.

Do you have any questions so far? OK. Let's begin.

I will now send you a link to our website, please open this link and make sure you are sharing your screen with us. Once you are in we will have you complete 4 tasks and rate the ease of use of the system on a scale from 1-10, with 1 being very hard and 10 being very easy. We will also ask you if you would prefer to use our website over your existing solution (TimeIQ) and if you would change something about the interface you just interacted with.

Part 2. Performing the Tasks and Providing Feedback

Now that you have access to our website, how would you go about

(Assume sign in information is; username: _____, password: _____)

Create a new rate card:

Please use the following data:

Title: 7Factor MQP Rate Card

Total roles: 3

Position / Rate: Senior Software Engineer / $150 - Junior Software Engineer / $86 - Intern / $65

How would you rate the ease of use of the system on a scale from 1-10, with 1 being very hard and 10 being very easy?

Create a new employee:

Please use the following data:

Employee Name: Jane Doe

Position: Senior Software Engineer

Default Rate Card: 7Factor MQP Rate Card

How would you rate the ease of use of the system on a scale from 1-10, with 1 being very hard and 10 being very easy?

Create a new project:

Please use the following data:

Project Name: “Staffing tool”

Client: “7Factor”

Rate Card: 7Factor MQP Rate Card

Budget: $100,000

Start Date: 4/1/2021

End Date: 10/10/2021

How would you rate the ease of use of the system on a scale from 1-10, with 1 being very hard and 10 being very easy?

Enter time:

Please enter the value “1” in all time log fields.

How would you rate the ease of use of the system on a scale from 1-10, with 1 being very hard and 10 being very easy?

Part 3. General Feedback and closing Remarks

Ok, you did it! Thanks for doing the test. Now we will wrap up with just a few quick questions about how the test went.

What is your overall impression of our application and how it compares to the existing system 7Factor uses?

Do you think the User Interface is lacking any particular feature or features?

Do you have any other suggestions about how we could improve our design?

Do you have any questions for me, now that we're done?

[End Script]

Test Metrics

Users who have never used our UI before will be asked on a scale of 1-10 to rate how easy it was for them to complete the task only with the bare minimum instructions given. Users who might ask a question might not get an answer back from an observer or facilitator right away in order to see their next course of action.