School of Transportation Burnaby Dept. App

(SoTBy)

Project Handoff Documentation

Group V5

Daniel (Bong Hyun) Chung

Sam Khademolhosseini

Timothy Tan

Maximilian Wu

# Table of Contents

[**Table of Contents**](#_gybidicucdjp) **2**

[**Overview**](#_icm5lnkv6xxc) **6**

[Schedule Timeline:](#_gjzk1t5s9k9v) 6

[Detailed Schedule:](#_3b9x21gvkmda) 6

[Vacation Request Form:](#_d6xshrj41spm) 6

[Vacation Approval Page:](#_rbezs6mkcbtq) 6

[Login:](#_oth1qigrc84q) 7

[Create User Page:](#_bz4389ti1pc1) 7

[Create Course Page:](#_ra2yus84yxkh) 7

[Create Resource Page:](#_ygn7h5wd4fc0) 7

[**Installation & Starting the app**](#_7tsmgbayvkn5) **7**

[Prerequisites:](#_ps7s7igblcth) 7

[Database Setup](#_2z5fp814nbdj) 7

[App Installation](#_qetgxepijehk) 8

[Starting the app](#_txyx1ep1sdbr) 8

[App usage](#_hc07in9yugio) 8

[Creating Users](#_81nuogespdyt) 8

[Creating Courses](#_pm2gawk9l9dd) 8

[Creating Resources](#_sw3niecuasjn) 9

[Updating Description on Detail Schedule Page](#_9rpvfecimkpd) 9

[Booking a Resource](#_6gghq7fd2wky) 9

[Submitting Vacation Request Forms](#_b1yqdllz5f46) 9

[Processing Vacation Requests](#_weh6bdyj2zj3) 9

[Notes](#_ipg722fajrtq) 10

[**Schedule Timeline**](#_dpdz2opp82ol) **11**

[Structure](#_tk74mug8gmf) 11

[Technical Details](#_9p00nm1hy1mz) 11

[DefaultNav](#_nm32fsl7wg4g) 11

[UserNav](#_5lust5ajwpiu) 11

[AdminNav](#_n2bi02rsan0l) 12

[Schedule](#_naanwhfupgc9) 12

[Timeline](#_8vc5y4pr6w17) 12

[Interactive Grid](#_87f8vifke6in) 13

[ReactGridLayout](#_g4u1dulwzeq4) 14

[TimelineGrid](#_i4blw3v4nath) 14

[RowHeader](#_2ydc9qcymvz1) 14

[Slot](#_4tsvmore1az2) 14

[**Detail Schedule Page**](#_ocm1fxly9e41) **15**

[Structure](#_318q9794lffx) 15

[Technical Details](#_dlzvg2wmf7bj) 15

[DetailedSchedule](#_ed188wwa06zx) 15

[Course](#_9egdj96lxoy8) 15

[Week](#_2930tqnkaa87) 16

[Day](#_9omwu499rfs2) 16

[**Login Page**](#_fips5ke1s7re) **17**

[Structure](#_4nt356lfr9aq) 17

[React Components](#_lsjx2fyizpwz) 17

[Header](#_zf3gqnbrbwkt) 17

[Technical Details](#_t1u298nk1g6s) 18

[Login](#_rg8fjwrply9c) 18

[**Create User Page**](#_5llfy9n50dgn) **18**

[Structure](#_rvpc6phgaznm) 18

[React Components](#_rgl3w42e00eg) 18

[Header](#_vfsjzh734pdx) 18

[Technical Details](#_d9mprflgtb2m) 18

[CreateUser](#_jb572zhekg09) 18

[**Create Course Page**](#_jft6tql2gj2d) **19**

[Structure](#_qniifuacy0x2) 19

[React Components](#_o8xrtx850ycl) 19

[Header](#_5i6f1ugp1gta) 19

[Technical Details](#_9hojmla6l2y3) 19

[CreateCourse](#_c1irfe7l4g1m) 19

[**Create Resource Page**](#_oa0mbpx8mheg) **19**

[Structure](#_2ymh5g2tlclo) 19

[React Components](#_thaxucjphhv0) 20

[Header](#_n434zhc2v6hp) 20

[Technical Details](#_12heft7936nk) 20

[CreateResource](#_kb9hnkh6mnza) 20

[**Resources Page**](#_5wb58shq3yaz) **20**

[Structure](#_xan2ae6ia4tk) 20

[Technical Details](#_3ug4cna6chz8) 21

[Resource](#_n0q33w6wrgw9) 21

[Resources](#_47baa11m83l6) 21

[Item](#_9x0s5yxqo7bw) 21

[PopUp](#_viyo4bvgo3zo) 22

[**Vacation Request Form**](#_aa3mojf18pvk) **22**

[Structure](#_r4t5vze5rpng) 22

[Structure General Comments:](#_99j1puyc826e) 23

[React Components](#_26p8804qry4i) 23

[GridDoubleLabel / GridNotesLabel / GridTripleLabel](#_yi8omhbrpugp) 23

[Header](#_sfypk1seotgv) 23

[UserInfo](#_oe30oyys00r2) 23

[VacationInput](#_6teyaaubeczj) 23

[Vacation](#_9b94t8ksj4hi) 23

[VacationList](#_x06boll6fss7) 24

[Technical Details](#_rtc5nbcr5je3) 24

[State](#_oifxnhgsy42e) 24

[Notes](#_p4xv06wk5nsy) 24

[**Vacation Approval Page**](#_gt2t8g93r7pd) **25**

[Structure](#_7lv8la88ma9i) 25

[Structure General Comments:](#_7pq1ufdapevv) 25

[React Components](#_emxo96xviiog) 25

[NoVacations](#_8rxk61a4kzce) 25

[VacationEntry](#_1kl64a912nmi) 25

[NotApprovedList](#_yulpsvfhxyuz) 26

[Technical Details](#_hixl6y1h0cuj) 26

[State](#_veigs9mc7d3e) 26

[Sockets](#_krx13wisnskf) 26

[**Server and Backend**](#_26fnnismldi) **27**

[Overview](#_xbrhx7m0o96q) 27

[Sockets](#_ka2ek98z8n3g) 27

[Overview](#_cikyjbe38hhm) 27

[Usage](#_xbxn0oclg7hr) 27

[Requests](#_n6jyeyxx7z1e) 28

[**Unfinished Features**](#_o35abudjv4sm) **30**

# Overview

The School of Transportation Burnaby Department App (SotBy) is an app designed to make scheduling of instructors and resources easier with the usage of live editing (sockets) and a database. In this document, technical details regarding this app will be explored and discussed for the purpose of making the project handoff as smooth as possible. Note that the front end and user interface will be unpolished due to our priorities being on the functionality of the app.

At its current state, the app uses React, PostgreSQL, Socket.io and Express. Pages available include a timeline page, a detailed schedule page, a login system and an admin page.

## Schedule Timeline:

The timeline page is the main page where instructors can view courses, users and vacations for the entire year.

## Detailed Schedule:

If a user double clicks on a course on the timeline, they will be redirected to the detailed schedule page. This page will display resources and instructors assigned to a course on a day-by-day basis.

## Vacation Request Form:

The vacation form submission page is where instructors can submit vacation requests. After submitting the form, the request shows up on the admin’s request approval page for processing. If the request is approved, it will appear on the schedule timeline page.

## Vacation Approval Page:

The vacation approval page is an admin-only page where admins can view the vacation requests made by users and either approve or reject them.

## Login:

The login page is where users can login as either an instructor or admin. Admins can perform more actions while instructors are limited to less.

## Create User Page:

The create user page is an admin-only page where admins can create new admin or instructor users.

## Create Course Page:

The create course page is an admin-only page where admins can create new courses to be assigned to instructors.

## Create Resource Page:

The create resource page is an admin-only page where admins can can new resources to be booked by instructors.

Refer to the table of contents to view each page in more depth.

# Installation & Starting the app

## Prerequisites:

Npm, postgresql/ pgAdmin, Windows

## Database Setup

1. Install the .sql file for creating the database
2. If you have pgAdmin 4 then:
   1. Create a database called ‘sotby’ by selecting Object > Create > Database
   2. Launch the query tool and import the .sql file
   3. Execute the .sql file
3. Otherwise If you just have postgres:
   1. Launch the command line terminal
   2. Ensure you have environment variables set up by typing `postgres -V`
   3. Create a database named `sotby` using the createdb command
   4. Execute the following line: `psql -U postgres -d sotby < C:\sql\_file\_path`
4. Check the tables have been created by navigating in the browser:

Servers > PostgreSQL 14 > Databases > sotby > Schemas > public > Tables

## App Installation

1. Clone the repository and project files locally
2. Type `npm install` in the `app` folder to install project dependencies
3. If your postgres admin password differs from ‘password123’ then:
   1. Navigate to App > Server > index.js and App > Server > requests.js
   2. In the Pool object, change the password field into your admin password
4. All done!

## Starting the app

1. Launch the command line terminal
2. Navigate to the server folder inside of the project’s app folder
3. Type `node index.js` or `npm start dev` in the terminal to start the server
4. Navigate back to the app folder
5. Type `npm start` or `npm start dev` in the terminal to start the front end

## App usage

### Creating Users

1. Login as an admin user to reveal the {Create User} option.
2. Click on {Create User} in the navigation bar.
3. Fill out the fields appropriately.
4. Click {Create User} to add the user to the database.

### Creating Courses

1. Login as an admin user to reveal the {Create Course} option.
2. Click on {Create Course} in the navigation bar.
3. Fill out the fields appropriately.
4. Click {Create Course} to add the course to the database.

### Creating Resources

1. Login as an admin user to reveal the {Create Resource} option.
2. Click on {Create Resource} in the navigation bar.
3. Fill out the fields appropriately.
4. Click {Create Resource} to add the resource to the database.

### Updating Description on Detail Schedule Page

1. Login to your account.
2. On the main timeline page, click {detailed} for the course you are trying to access.
3. Find the date you want to edit.
4. Click {edit} to make the description editable.
5. Enter a new description and click {save}.

### Booking a Resource

1. Login to your account (to be able to go to the detailSchedule page).
2. On the main timeline page, click {detailed} for the course you are trying to book a resource for.
3. Find the date you want to book a resource for.
4. Click on {book} under the resources column.
5. Find the resource you want to book.
6. Click on {book} again.
7. Enter the number you want to book and click {submit}.

### Submitting Vacation Request Forms

1. Login to your account to reveal the {Request Vacation} option.
2. Click on {Request Vacation} in the navigation bar.
3. Follow the labels and input your information.
4. Fill out the vacation section and click add for every request you want to make.
5. Click submit to submit the request.

### Processing Vacation Requests

1. Login to an admin account to reveal the {Approve Vacation} option.
2. Click on {Approve Vacation} in the navigation bar.
3. Click {Approve} to approve a request or click {Reject} to reject a request.
4. Click {Back} at the top of the page to go back to the main schedule timeline.

## Notes

* The package.json files can be configured to start both the server and app with just one `npm start` command but will be left as is for separate development of both server and front end
* “npm start dev” restarts the server anytimes there’s a change using nodemon

# Schedule Timeline

## Structure

The timeline is structured in the following components

Schedule

* Timeline
  + DefaultNav
  + UserNav
  + AdminNav
  + Month
  + InteractiveGrid
    - ReactGridLayout
    - TimelineGrid
      * RowHeader
        + Slot

**Structure General Comments:**

* The form component was used in development to create courses and users but will not be listed
* Sockets are passed down the hierarchy to allow for events sent to and from the backend to change the state of components. The components that need sockets are InteractiveGrid and TimelineGrid.

## Technical Details

### DefaultNav

The DefaultNav component displays a navigation bar to all users, allowing them to login to their account.

### UserNav

The UserNav component displays a navigation bar to instructors who have successfully logged in. Here they are able to access the vacation request form or logout of their account.

### AdminNav

The AdminNav component displays a navigation bar to admin users who have successfully logged in. They are able to access the following features:

* Create user
* Create course
* Create resource
* Request vacation
* Approve vacation
* Logout

### Schedule

The schedule component starts the app and makes a request to the server to retrieve the list of instructors. It then passes its data to the Timeline component. Note that this page will also display a modal for errors occurring from the other pages.

### Timeline

The timeline component creates the UI for the months with a css grid and passes the instructor data to the InteractiveGrid component.

Note the following functions:

getWeeks:

* Retrieve the number of weeks in a month and store it into a “weeks” array
* This array will be used in creating grid columns for each month

getNumberOfWeeks:

* This function will take the weeks array and will output the starting x position the month should be created at in the css grid
* When creating an instance of the Month component, the x positions are offset by 3 since the first 2 columns will be used for displaying instructors

Timeline General Comments:

* At the moment, it is hardcoded to only calculate the months for 2022. The front end should take in code to change the year

### Interactive Grid

The interactive grid component allows course elements on the timeline to be added, deleted and moved.

It uses a React library called “react-grid-layout” and its ReactGridLayout component for dragging course elements to other columns. It also contains many functions and socket method calls used for removing and adding course elements and users.

Multiple times throughout the document, lodash (which is “\_” in the imports) will be used for modifying arrays for updating the state of the component.

Its state has the following attributes:

items:

* A list of objects with the following attributes
  + key:
    - The key of the item used for uniquely identifying that element in the tree
  + name:
    - First and last name of the user which will be displayed
  + timeblocks:
    - List of objects which contains information for displaying courses for a specific user
* Each timeblock object will have the following attributes:
  + courseNum:
    - The course number uniquely identifying a course
  + start:
    - A date object that contains the start time for the course
  + end:
    - A date object that contains the end time for the course
  + name:
    - The name of the course which would be displayed
  + userId:
    - The id of the user a course belongs to

WeekInformation

* An object passed in from the Timeline component used for mapping the dates of courses to an index in the grid

HeightLimit

* An integer indicating the max row that elements can be dragged to
* Currently unused since every course element movement is restricted to its row only

### ReactGridLayout

The ReactGridLayout component is a component from the “react-grid-layout” library. This component uses the gridstyles.css file to style its grid elements.

You can take a look at its documentation here: <https://github.com/react-grid-layout/react-grid-layout>

Notable component settings:

* allowOverlap:true
  + Allow elements to be created on top of each other
  + Prevents elements from pushing other elements onto different rows
* resizeHandles: [‘e’]
  + Creates a handle for resizing a course element to the right
  + Currently, including the handle for resizing to the left is the same as its right counterpart so it isn’t used

### TimelineGrid

The TimelineGrid component handles creating and removing users. Rowheader components are created from each user passed into this component from the InteractiveGrid component.

### RowHeader

The RowHeader component consists of 2 rows in the grid and it creates a Slot component for every grid cell.

### Slot

The slot component consists of a button to create a course that will be placed at that specific slot’s position. It calls the createCourse function passed from the RowHeader which creates the element in the InteractiveGrid component.

# Detail Schedule Page

## Structure

The detail schedule page is structured in the following components:

DetailedSchedule

* Course
  + Week
    - Day

**Structure General Comments:**

* Sockets are passed down the hierarchy to allow for events sent to and from the backend to change the state of components. The components that need sockets are Day and Week.

## Technical Details

### DetailedSchedule

DetailedSchedule is a function that calls the Course component and passes socket and courseNum to it. It takes the socket as an argument, and gets courseNum from the url.

### Course

Course represents a complete course from the beginning to its end. All data related to this course is retrieved in this component. The raw data is basically a list containing information about every day for this course.

* **retrieveDailyScheduleDataFromDatabase:**
  + Sends a request to the server to get the raw data for the course we want to display on page. If successful, we send the raw data to the parseData function.
* **parseData:** 
  + This function formats and stores the data received from retrieveDailyScheduleDataFromDatabase() in the state of Course.
  + Main part of this function is the second for loop. The raw data received has duplicated rows if there are more than one resource booked for a single day.  
    After this loop, each item in formatted data will have the info for a single day, and if there are multiple lists for a day, it is stored in a nested list in it.
* **renderCourse:**
  + The data stored in the state is a list of day info. In this function, we divide that list into many chunks, each chunk having data for a week or 7 days. Then it passes data for each week to the renderWeek function.
* **renderWeek:**
  + Calls the Week component and passes the required info to it. Note that we need to pass a key to it since we used the map method in renderCourse. The key for each week element is the ds\_id of its first day.

### Week

In simple words, a week is a table holding information for 7 days. It stores the info for the 7 days in its state, so if a change occurs, it re-renders that part of the page.

* **render:**
  + Sets up the table and table heading.
  + Using map function again, we pass the info for a single day to renderDay.
  + A day would be a single row in our table.
* **renderDay:**
  + Calls Day component and passes required info to it. Once again, we need to pass a key since we used the map method to render it.
* **this.socket.on('changeDay', (rowInfo)):**
  + Gets invoked with ‘changeDay’ signal. Then, it checks to see the info of which day has been changed based on ds\_id. Then, replaces the old info for that day with the new info. (rowInfo is the new info)
  + Updates the state -> re-render the part of the page that’s been changed.

### Day

Represents a single day. It can be thought of as a single table row.

* **render:**
  + Date and instructor are always uneditable.
  + Description can be changed once we click on the edit button in the last column. The value of this EditText is stored in the state of Day and gets updated every time we change it. (handleSave)
  + Last column holds a button that either makes the description editable or saves the changes after we have edited the description. (handleEditSave)
  + Resources:
    - column displays all the booked resources for that day and also a book button that redirects the users to resources page.
    - If you remember in the parseData function in Course, we added all booked resources to a list. Therefore, we use the map methods on that list to list all the resources. The key of each list item is the model\_num of the resource.
* **handleEditSave:**
  + Updates the state of the edit/save button.
  + If the value of editMode is false, it means the button is displaying “Edit” and the description is uneditable.
  + If it’s true, then we assume, user is currently making changes to descriptions. And the button displays “Save”. Once they click on save, we send a signal through sockets, to:
    - update the database.
    - update state in week and to hopefully re-render the updated section for all users.

# Login Page

The login page is where admin users navigate to when they want to login to their account.

## Structure

The create course page is structured in the following components:

Login

* Header

**Structure General Comments:**

* Login uses a fetch request to verify if the username exists and then verifies if the input password matches the hashed password in the database.
* Upon successful login, the user is redirected to the Timeline page.

## React Components

### Header

The header component has the form title and back button that leads back to the schedule timeline.

## Technical Details

### Login

* Users will input their username and password in order to log in.
* When a user successfully logs in, their username, first name, last name, and admin status are saved in localStorage for their session (see Unfinished Features below).
* When a login fails, users will be met with a message stating that they have input an incorrect username or password.

# Create User Page

The create user page is where admin users navigate to when they want to create a new admin or instructor user.

## Structure

The create user page is structured in the following components:

CreateUser

* Header

**Structure General Comments:**

* CreateUser uses sockets to post to the database.
* When a user is added to the database, the Timeline page is re-rendered to display the new user.

## React Components

### Header

The header component has the form title and back button that leads back to the schedule timeline.

## Technical Details

### CreateUser

* Admin users will input the new user’s username, first name, last name, hiring date, admin status, email, and password here.
* Passwords are hashed before being added to the database using Argon2.

# Create Course Page

The create course page is where admin users navigate to when they want to create a new course to be assigned to instructors.

## Structure

The create course page is structured in the following components:

CreateCourse

* Header

**Structure General Comments:**

* CreateCourse uses sockets to post to the database.
* When a course is added to the database, admin users can then assign users to this course on the Timeline page.

## React Components

### Header

The header component has the form title and back button that leads back to the schedule timeline.

## Technical Details

### CreateCourse

Admin users will input the new course’s course number (CRN), subject, course, title, start date, end date, and the colour they wish to assign to the course.

# Create Resource Page

The create resource page is where admin users navigate to when they want to create a new resource for instructors to book.

## Structure

The create resource page is structured in the following components:

CreateResource

* Header

**Structure General Comments:**

* CreateResource uses sockets to post to the database.
* When a resource is added to the database, instructor users can book available resources for the day from the DetailedSchedule page .

## React Components

### Header

The header component has the form title and back button that leads back to the schedule timeline.

## Technical Details

### CreateResource

* Admin users will input the new resource’s model number, model name, total quantity, and the location of the resource.
* The quantity remaining will automatically be set to the total quantity in the database.

# Resources Page

In the resources page, users can see all the resources available for a day, and book them, if there is any left.

## Structure

The resources page is structured in the following components:

Resources

* Resource
  + Item
    - PopUp

**Structure General Comments:**

* Sockets are passed down the hierarchy to allow for events sent to and from the backend to change the state of components. The components that need sockets are Day and Week.

## Technical Details

### Resource

Resource is a function that calls the Resources component and passes socket, date, and ds\_id to it. It takes the socket as an argument, and gets data and ds\_id from the url.

### Resources

Resources component is the component that contains the main structure of our page and the data.

* **retrieveDailyScheduleDataFromDatabase:**
  + Sends a request to the server to get the information about the resources. If successful, we send this raw data to the parseData function.
* **parseData:**
  + It parses the raw data. Then, it stores it in the state of Resources, and sets dataLoaded to true.
* **renderResouce:**
  + Sets up the table and table heading.
  + Using the map methods, passes the data to the renderItem function. And uses the result as the table body. (Items are a bunch of table rows)
* **renderItem:**
  + Calls the Item component and passes the required info to it. Note that we need to pass a key to it since we used the map method in renderResource. The key for each Item element is the combination of model\_num and the quantity left of that resource.
* **this.socket.on('bookResource', (resInfo)):**
  + When signaled, it finds the info for which resource has been changed by comparing the resInfo’s model\_num with model\_num of all the resources we currently have.
  + When it finds it, it replaces its info with the new info received from resInfo, and updates the state.
  + Updating the state causes a re-render and we see the new values for the updated resource without needing to refresh the page.

### Item

Represents a single resource. It can be thought of as a single table row.

* **render:**
  + The interesting part is the last column (last td tag). We either display a button that says ‘Book’, or display the popup component based on the value stored in the state (seen).
* **togglePop:**
  + Simply updates the state.
  + Note that this function is passed to the PopUp components.
* **this.resInfo.q\_left (variable in constructor):**
  + If we have a value for it, we keep it. However if it’s null, it means that we have not yet booked any of this resource for this date. Therefore, the quantity left (q\_left) would be equal to the total quantity of the resource.

### PopUp

Not really a popup to be honest. (The goal was to make it a popup)

It’s basically a form that asks users how many of a resource they want to book. Then, it processes the request.

* **render:**
  + Simple form.
  + Calls updateInputValue when something is typed in the input field, and calls handleSubmit when submitted.
* **updateInputValue:**
  + Stores current value of the input field by storing in (updating the) state.
* **handleSubmit:**
  + Checks if the input is valid or not.
  + If valid, sends a signal through sockets to:
    - update the values in the database.
    - update the state of Resource, to hopefully re-render the page and display the new values.

# Vacation Request Form

The vacation request form page is the page where users navigate to when they want to submit a vacation request.

## Structure

The vacation form is structured in the following components:

Vacation

* Header
* UserInfo
* Labels
* VacationInput
* VacationList
  + Vacation

### Structure General Comments:

* Vacation submission uses sockets instead of fetch requests to post to database
* If there are no vacations added yet, a message is displayed indicating so.
* If there are vacation(s) added, the VacationList component is displayed with a Vacation component for every vacation.

## React Components

### GridDoubleLabel / GridNotesLabel / GridTripleLabel

These are just labels formatted in different ways using flexbox.

### Header

The header component has the form title and back button that leads back to the schedule timeline.

### UserInfo

Users input their first name, last name, employee ID, and extension here. This is used to determine who sent the vacation request. The submit button is clicked to submit the request.

### VacationInput

Users input information about their vacation request here. The labels are used as a guideline so users know what to put for the input boxes.

### Vacation

After the user completes a vacation request and clicks add, a vacation component is created under the form with the vacation request information.

### VacationList

A list of vacation components. Every vacation request the user fills out is made into a vacation component and added to this list. Users can see all the requests before clicking submit.

## Technical Details

### State

* user:
  + the first and last name are taken from the input field and converted to a username format (first\_last) eg. john\_doe
* users:
  + before the component is loaded, a list of users is grabbed from the database
  + if the first and last name the user inputs is not in the list, they can not submit a request
  + only users that are registered in the database can submit a request
* vacations:
  + every vacation the user adds is added to this list
* vacationID:
  + id used to remove vacations from the vacationList component
  + this is not the same id that is stored in the database
* vacationSubmitted:
  + boolean used to display submission status
  + true if the submission is successfully posted
* loaded:
  + boolean used as a buffer for getting users
  + true once the list of users is retrieved from the database

## Notes

The following fields aren’t currently used:

* EmployeeID
* Extension
* N-New/C-Cancel
* Hours
* VA-Vacation/BT-Banked Time
* Notes

These fields are implemented in the form anyways to allow for the possibility of its usage in future releases.

# Vacation Approval Page

The vacation approval page is an admin-only page where admins can view the vacation requests made by users and either approve or reject them.

## Structure

The vacation approval page is structured in the following components:

Vacation

* Back Button
* NoVacations
* NotApprovedList
  + VacationEntry

### Structure General Comments:

* If there are no vacations for the admin to process, the NoVacations component is displayed.
* If there are vacations for the admin to process, the NotApprovedList component is displayed instead with a VacationEntry component for every vacation.

## React Components

### NoVacations

A simple display message indicating that there are no vacations for the admin to process this time.

### VacationEntry

Every vacationEntry has the following information:

* name
* start date
* end date
* duration

Along with this information are two buttons:

* Approve
  + The approved column for this entry in the database goes from 0 to 1. The schedule timeline will display all vacations approved.
* Reject
  + The vacation is deleted from the database.

### NotApprovedList

The NotApprovedList component consists of VacationEntry child component(s). If a VacationEntry component is processed, it is removed from the list.

### 

## Technical Details

### State

* vacations:
  + before the component is loaded, a list of unprocessed vacations is retrieved from the database
  + processing a vacation entry removes the vacation from this list
* loaded:
  + boolean used as a buffer for getting vacations
  + true once the list of users is retrieved from the database

### Sockets

Seeing that there can be more than one admin processing requests at any given time, sockets are used for live editing. Processing a vacation sends a signal to the server to perform the put/delete request. The server then emits another signal to all the clients, sending the changes and having the client rerender that particular component. In other words, admins would not need to worry about processing the same request. Any changes made to the page are updated in real time.

# Server and Backend

## Overview

The backend handles updating the database and broadcasting events from each client to all the other clients. Technologies used in implementing the backend are Node.js, Express.js, Socket.io and PostgreSQL. Currently, it consists of 3 files inside of the server folder.

Index.js

* Starts the socket, server and database connection
* Contains routes

Socket.js

* Contains event listeners for emitting events to all clients connected to the socket
* Updates the database using queries from requests.js

Requests.js

* Contains functions that return promises for querying to the database.

## Sockets

### Overview

Sockets (WebSockets) provide a full-duplex communication channel over a single TCP connection. This allows for bi-directional communication between the server and app. They are used for the live-edit feature of this app and allows for multiple users to modify data concurrently.

Socket.io, a library built upon websockets, is currently being used in this app and you can view its documentation here: <https://socket.io/docs/v4/>.

### Usage

Sockets are currently used to update the database as well as notify clients of changes. To communicate between the app and the server, event listeners and event emitters will need to be used on both ends.

The server’s event listeners are structured in the following:

socket.on('event-type', (data) => {

databaseQueryPromise()

.then(response => {

socket.broadcast.emit(‘event-type’, data)

})

.catch(error => {

socket.emit(‘error’, error)

});

})

After the client sends an event over to the server, the socket matches the event type with the event sent. The database then calls a function that returns a promise which queries the database.

If the query was successful, the socket’s broadcast attribute is used to send the data to all clients other than the one that emitted the event. If unsuccessful, an error event is emitted to the client who sent the message to indicate that their change was unsuccessful.

Errors that may occur would be any database error related to constraints or data input.

The client’s event listeners are structured in the following:

this.socket.on("event-type", (data) => {

this.frontEndUpdateFunction(data, false);

});

After the server emits an event to clients, the clients receive the data through this event listener. They would then call a function to update the front end or state in some way.

One thing to note would be that the front end update function may include a default parameter called ‘emit’ set to true. This attribute dictates whether or not to send the data to the backend so that all other clients will receive the data.

In this case, the client updates the front end immediately through this update function then sends data to the backend so that all the other connected clients can call this update function. The other clients however, would call the update function with the emit argument set to false so that it doesn’t re-broadcast the message and cause an infinite loop.

## Requests

The requests.js file contains functions that return promises for querying the database. These functions are exported and used in the socket.js and index.js files.

## 

# Unfinished Features

* Reporting feature
* Recently changed notification
* UI
  + Course coloring
  + Element styling
  + Grid styling
  + Admin page
  + Animations
  + Mobile friendly
* Creating correct grids for years other than 2022
* Processing a vacation request should notify the user with a comment
* Remote hosting