

Calendar+ Final Report

Authors: Chad Kinnard, Bo Yang Li, Tyler Marrapese, Warren Overstreet

Introduction

For many of us, time is the most precious resource as it can never be returned. As such, time budgeting has become one of the most important tools in our daily lives. It is not only about knowing what one might do at what time and for how long, but also about other important details such as the specific details of planned events and the location of such events. Along the way, one might even desire to be reminded of important events some time before the event starts. Indeed, a calendar application would promote personal accountability, realistic event/task duration estimation, task prioritization, and event time boundary setting.

Introducing Calendar+. Calendar+ is a calendar run as a dynamic web application that helps users personalize their schedule as they see fit. Through a simple account creation, a user's calendar is personalized to them. Accessed through the Google Chrome web browser, users can view their calendar from any Windows 10 device. Calendar+ makes it to where users can plan their days, weeks, months, and years according to their own needs. Through web browser notifications, users are reminded of important events that they have scheduled.

Time management is an important part of day-to-day- life, whether as a student, or a full-time employee, and Calendar+ acts as a place for users to keep track of their schedules and become more productive.

Definitions

Presentation layer/Frontend - The part of the web application seen and interacted with by the user.

Backend - The part of the web application not accessible by the user; typically contains the web server, API, and database.

API - Application Programming Interface. It is a group of functions and procedures that allows the exchange of data between applications.

XAMPP - Open source web stack. Stands for "Cross platform Apache MariaDB Perl PHP."

Relational Database (RDBMS) - Database based on relations between data.

Node.js - Open Source cross-platform backend JavaScript runtime environment
 npm - Node.js package manager

Express.js - Node.js web application framework

Middleware - Functions that have access to the request and response objects, and that can call the next middleware function in the stack.

Request-Response Cycle - How user requests are handled in a web application.

Version Control System - Keeps track of and controls changes in a project, contains a full history of the project.

DOM - Document Object Model. Represents the structure of a document, connecting the HTML, CSS, and JavaScript together.

AJAX - Asynchronous JavaScript ~~and XML~~.

Event Listener - Function that executes when a specified event occurs, such as a mouse click or an internal timer.

Technology

Calendar+ is developed using HTML, CSS, and JavaScript as its programming languages and Visual Studio Code as its code editor. Targeted towards the Google Chrome browser on the Windows 10 operating system, Calendar+ utilizes Node.js as the backend JavaScript runtime environment, Express.js as the web application framework, MySQL via XAMPP as the database, and Linode as the cloud hosting provider. In addition to Express.js, Calendar+ uses the following Node packages and Express.js middleware: morgan, mysql, body-parser, dotenv, nodemon, web-push, and express-session. Finally, the version control system for Calendar+ is git and the code repository is GitHub.

Our team chose a JavaScript-based stack mainly to simplify the project and reduce learning requirements. Had we chosen a different language such as PHP, C#, or Python for our backend, we would have needed to spend more time learning different languages. By using JavaScript in much of our project, we only needed to concern ourselves with reading API documentation.

As one of the most popular backend options, Node.js/Express.js has an incredibly rich ecosystem that provides us with any module we would need and dozens of community boards (i.e. Stack Overflow) that could help us solve any coding issue encountered. Indeed, we use morgan to keep track of HTTP requests, mysql to provide a database driver to the MySQL database, body-parser to process form information, dotenv to provide environmental variables, nodemon (dev dependency) to automatically restart node after saving a change without being required to type npm commands, web-push to support notifications, and express-session to save user session data as a server cookie. Express.js was chosen as it made backend development much easier compared to using the Node.js HTTP module.

We chose MySQL as our database mainly due to Warren's pre-existing knowledge about SQL and due to Dr. Nicholson's recommendation of XAMPP which comes with a MariaDB (MySQL fork) database. Also, relational databases made more sense as our data points are heavily related to one another. Using a NoSQL database such as MongoDB would have raised learning requirements and would not have been well suited to data that is not non-structured. We also chose Linode as our cloud host mainly due to Dr. Nicholson's familiarity with the platform. If not for Dr. Nicholson, there would be a world in which our team chose MySQL server and Heroku instead.

We chose git as our version control system as it integrates well with other tools as the predominant version control system, and because Bo had experience in it. As a distributed version control system, it also enables every team member to have their own set of the full history of the project without having to maintain a network connection with a central repository. This enables independent work since we work at different times outside the classroom. GitHub was chosen as our repository as it integrates well with git and Linode as the most popular online code repository; changes to the GitHub repository can be easily deployed to the Linode server.

We chose Visual Studio Code as our code editor because it was a familiar tool for our team. We also chose our target platform to be Google Chrome on Windows 10 as it is the most popular at the time of this writing. While we cannot guarantee it, Calendar+ would also function on any Chromium-based browser such as Edge.

Our decision to limit the number of technologies in our web stack enabled us to experience a much easier learning curve. We were able to come to terms with Node.js and Express.js without too much difficulty. Making things easier, our team had preexisting knowledge of HTML/CSS/JavaScript. Linode, git, and GitHub also were not difficult to learn with enough time. git commands were not even required most of the

time as we could simply git functionality in a GUI format via either Visual Studio Code or GitHub Desktop.

As a result, the technologies we chose and learned could be used effectively to solve our problems throughout the development process. Existing knowledge of vanilla JavaScript enabled DOM manipulation and AJAX commands, the ease of using git via GUI enabled easy and frequent commits, and the relatively easy learning process for server-side JavaScript enabled us to establish a functioning backend in reasonable time.

Design

UI / UX

The UI is implemented using HTML, CSS, and JavaScript. Most elements that appear on the website use CSS's **flex** property in order to maintain some level of scalability across different desktop resolutions.

Login Page

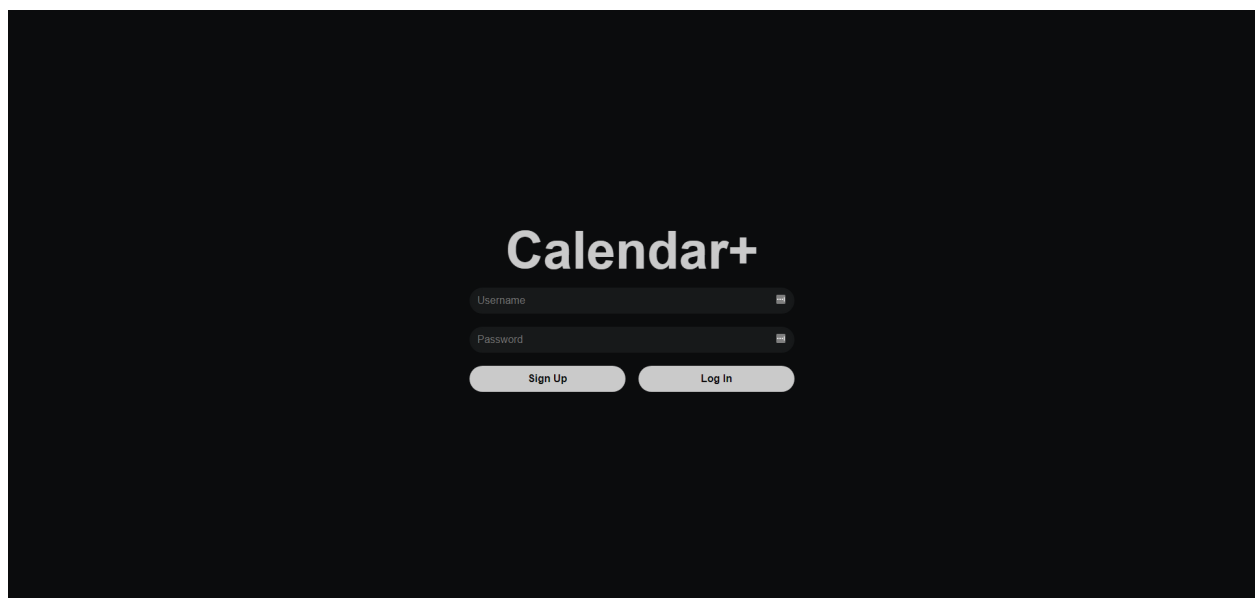


Figure 1 - Login Page

The login page consists of a “logo” for Calendar+ (just the name spelled out), two form fields, and two buttons. The two form fields are for the user to enter their Username and Password, while the two buttons will either redirect the user to the registration page or log them in, leading them to the month view.

Registration Page

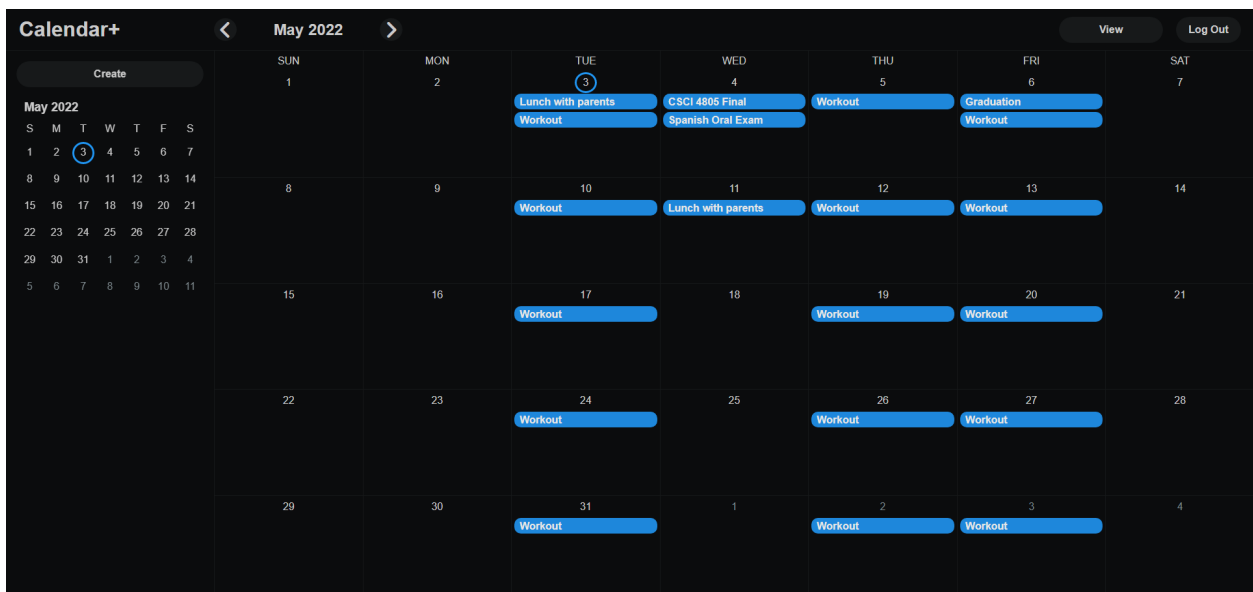


The screenshot shows the 'Calendar+' registration page. It features a dark background with white text. The title 'Calendar+' is centered at the top. Below it are four input fields: 'Username', 'Email', 'Password', and 'Confirm Password'. Each field has a small icon on the right (a calendar icon for Username, an envelope for Email, and a key for Password and Confirm Password). At the bottom are two buttons: 'Back' and 'Sign Up'.

Figure 2 - Registration Page

When the “Sign Up” button on the login page is clicked, it will open this view. It consists of four form fields, and two buttons. The four form fields are for the user’s Username, Email, Password, and an extra field for them to confirm their Password. The “Back” button will redirect the user back to the login page, while the “Sign Up” button will check the form for invalid fields, and if all fields are valid, submit the form (registering their account) and redirect the user back to the login page.

Month Page



The screenshot shows the 'Calendar+' month view for May 2022. The interface includes a 'Create' button on the left, a calendar grid for May 2022, and a 'View' button on the right. The calendar grid shows days of the week (SUN, MON, TUE, WED, THU, FRI, SAT) and dates. Events are listed for each day, such as 'Lunch with parents', 'GSCI 4805 Final', 'Spanish Oral Exam', 'Workout', and 'Graduation'. The date 3 is highlighted with a blue circle.

Figure 3 - Month Page

The month view, pictured above, is a 7x5 or 7x6 grid of cells containing the number of the day, and any events (up to three) that fall on that day. Clicking a date on the calendar brings the user to the day view for that specific date. Clicking the arrows at the top of the screen will change the calendar to the previous or next month, respectively. Clicking one of the event nodes will bring up the event on the sidebar, ready to be edited or deleted. Or, the user can click inside the empty space of one of the cells, and the sidebar will be updated to create a new event at that specific date.

Day Page

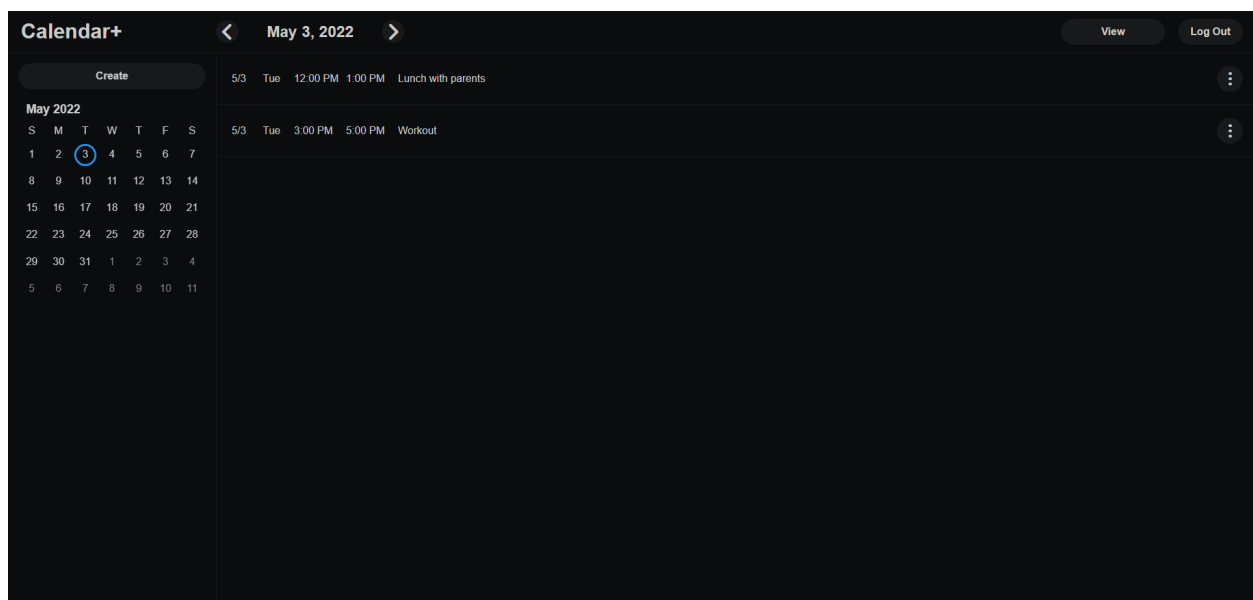


Figure 4 - Day Page

The day view, pictured above, is a vertical listing of all events which occur over the course of the currently selected day. Clicking the arrows at the top of the screen will change the view to the previous or next day, respectively. The three dots on the far right side of each event's container can be clicked to bring up the event on the sidebar, ready to be edited or deleted.

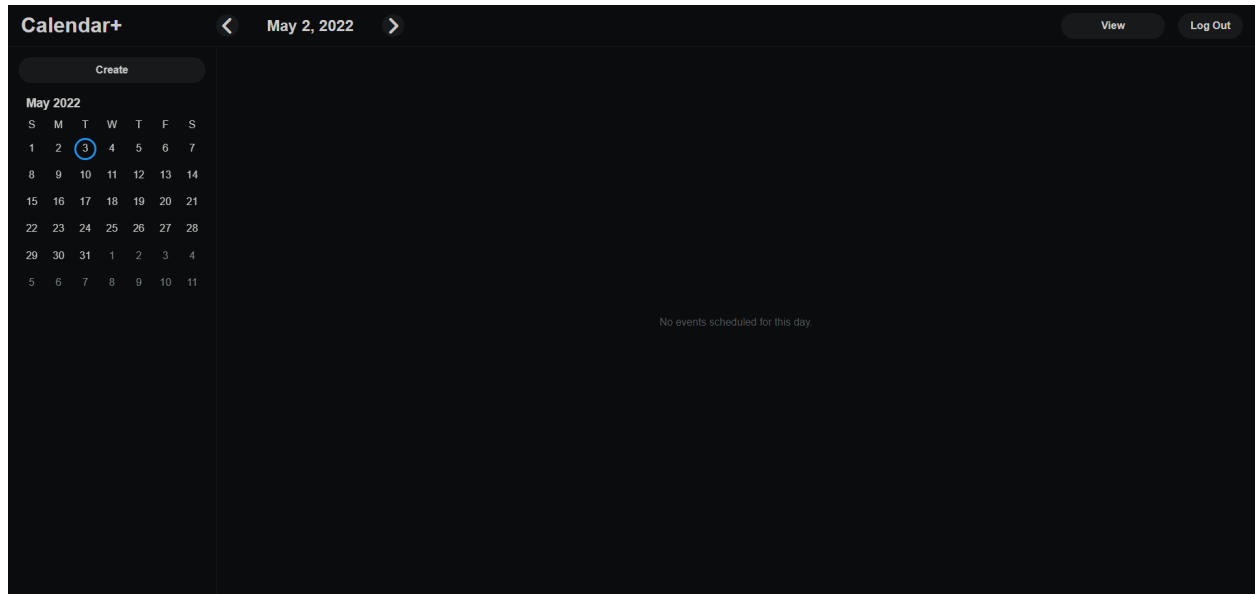


Figure 5 - Day Page with no events

Pictured above is an example of what the day view would look like if no events were scheduled for that day. It will simply read in the middle of the screen, “No events scheduled for this day.”

Week Page

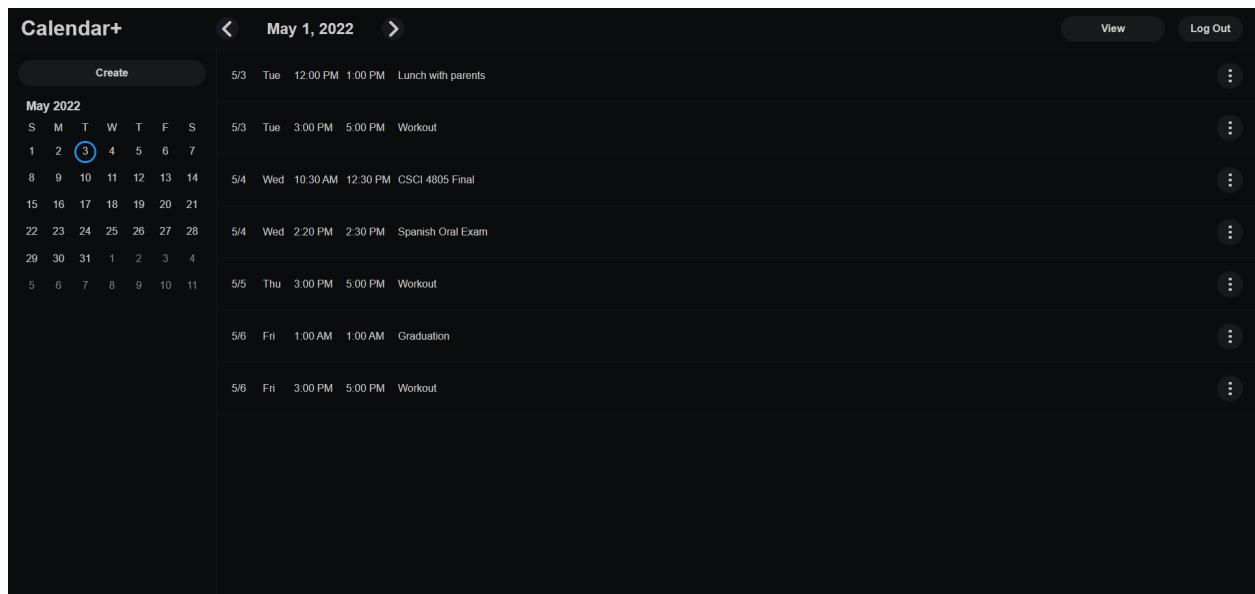


Figure 6 - Week Page

The week view, pictured above, is a vertical listing of all events which occur over the course of the currently selected week. Clicking the arrows at the top of the screen

will change the calendar to the previous or next week, respectively. Clicking the date (furthest left bit of text in the event's container) will bring the user to the day page for that date. The three dots on the far right side of each event's container can be clicked to bring up the event on the sidebar, ready to be edited. If no events are scheduled for that day, the middle of the screen will read "No events scheduled for this week." similar to the day page.

Year Page

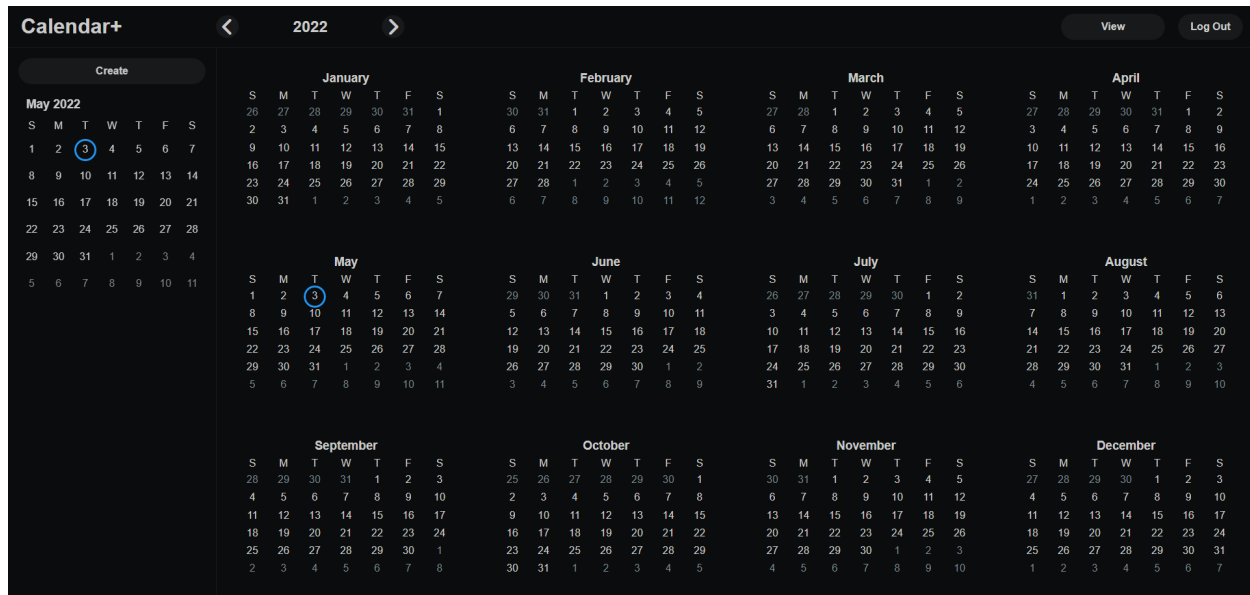


Figure 7 - Year Page

Schedule Page

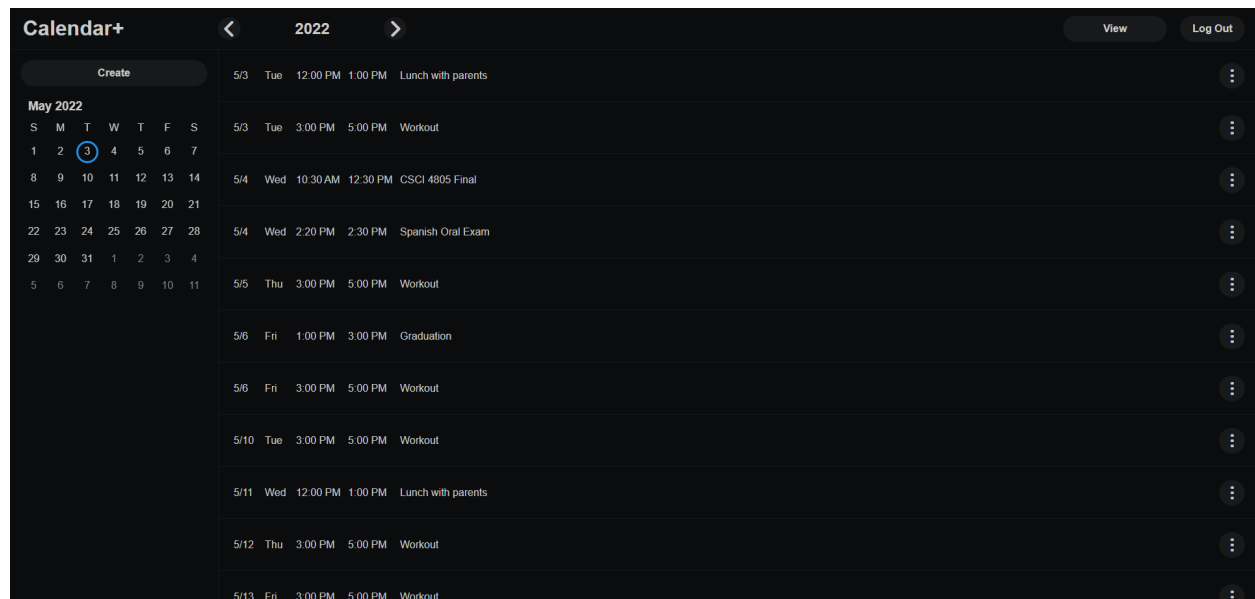
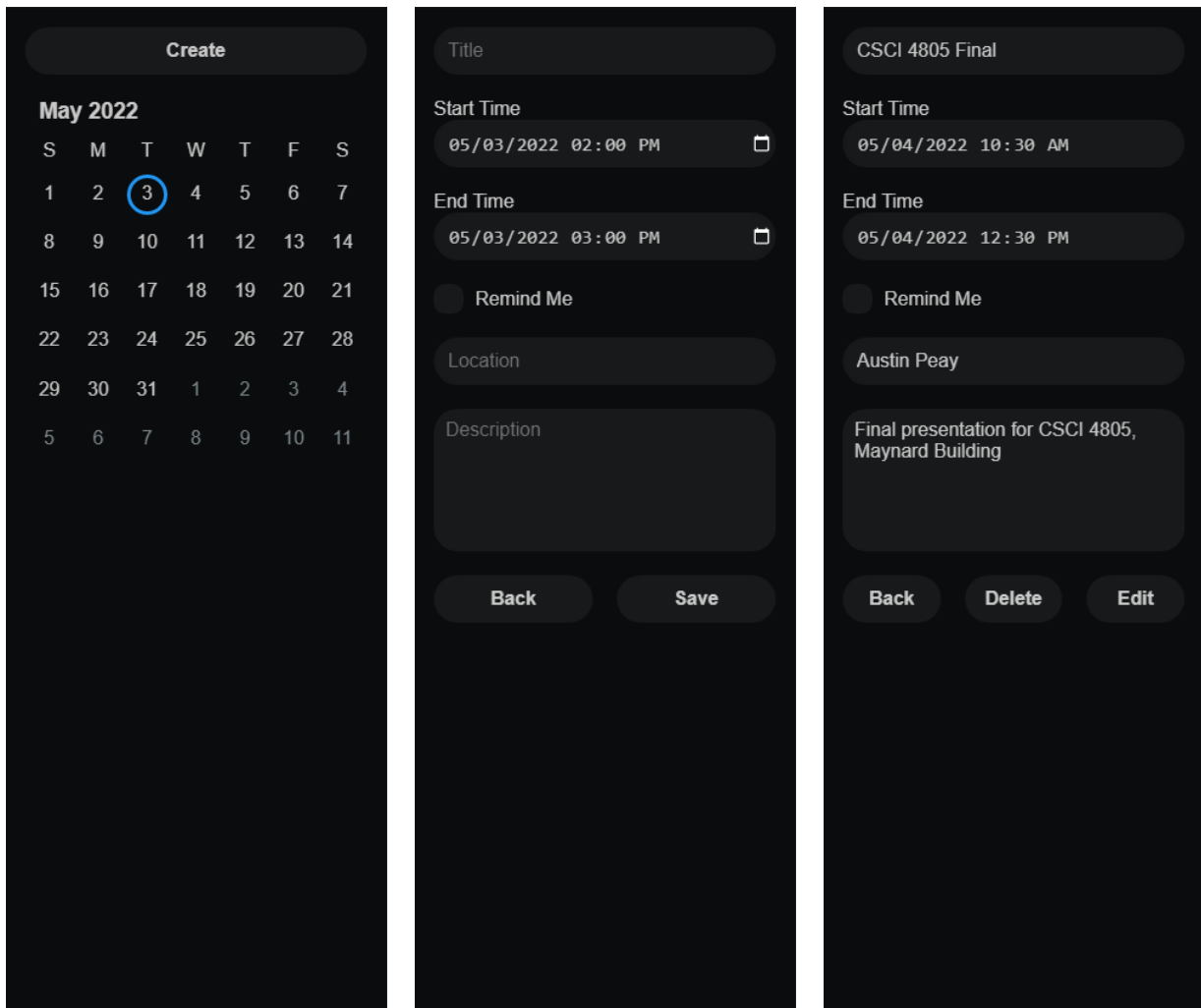


Figure 8 - Schedule Page

The schedule view, pictured above, is a vertical listing of all events which occur over the course of the currently selected year. Clicking the arrows at the top of the screen will change the calendar to the previous or next year, respectively. Clicking the date (furthest left bit of text in the event's container) will bring the user to the day page for that date. The three dots on the far right side of each event's container can be clicked to bring up the event on the sidebar, ready to be edited. If no events are scheduled for that day, the middle of the screen will read "No events scheduled for this year." similar to the day and week pages.

Sidebars



Figures 9-11 - Sidebar Views

The left image pictured above is the main sidebar view, which is the default sidebar present on all pages. It consists of a button, which when clicked, will open the event creation sidebar (middle image). Under that, is a minified month-view calendar, which only shows the current month for easy navigation when the user is in a different view.

The middle image pictured above is the event creation sidebar view, which can be opened in a few ways, differing based on the current page, but can consistently be opened by clicking the create button on the main sidebar. It is a form consisting of 5 fields and a checkbox for reminders.

The right image pictured above is the event editing view. Clicking on an event in any of the calendar views will bring this up, with the correct event information pre-applied to non-editable fields. Clicking the delete button will delete the event, while

the edit button will “unlock” the form’s fields and allow the user to edit the event. The edit button will also change the buttons at the bottom to perfectly match the event creation view, so the delete button will disappear and will be replaced by a save button.

Scripts

calendar-common.js

Global Variables

The calendar-common.js file contains two general variables: **const currDate** and **const weekdaysAbbr**. The **currDate** variable is a Date object which is created using the default constructor, resulting in the current date used on the user’s machine. The **weekdaysAbbr** variable is simply an array consisting of all the abbreviations for weekdays (M, T, W, etc.).

renderDaysOfWeek()

Objective

Renders the days of the week (S, M, T, W, T, F, S) to the top of the calendar (is only used for the mini calendars)

Arguments

The renderDaysOfWeek() function accepts 1 argument: **cell**. “**cell**” is a reference to the div in which the weekdays should be rendered.

Operations

The function consists of a for loop with 7 iterations. In each iteration, a new cell containing the corresponding entry from **weekdaysAbbr[]** is created and appended through JavaScript DOM elements.

renderStartPadding()

Objective

Renders the days of the previous month that would be visible on the calendar for the current month.

Arguments

The renderStartPadding() function accepts 6 arguments: **cell**, **paddingDays**, **lastDayPrevMonth**, **month**, **year**, and **renderEvents**. “**cell**” is a reference to the div in which the cells should be rendered. “**paddingDays**” is the number of cells to render.

“lastDayPrevMonth” is the highest number to be shown inside one of the cells.

“month” is the number of the month. **“year”** is the full year in XXXX format.

“renderEvents” is a boolean which defaults to false.

Operations

The function consists of a for loop which creates and appends the cells through JavaScript DOM elements. If **“renderEvents”** is set to true, it also creates a div with the ID of the current date to be rendered. Following this, it adds an event listener to the div, which when clicked, clears the sidebar and opens it to the create view with that date pre-applied to the start and end times.

renderMonth()

Objective

Renders all the days in the selected month.

Arguments

The renderMonth() function accepts 6 arguments: **cell**, **daysInMonth**, **dt**, **month**, **year**, and **renderEvents**. **“cell”** is a reference to the div in which the cells should be rendered. **“daysInMonth”** is the number of cells to render. **“dt”** is a reference to a Date object. **“month”** is the number of the month. **“year”** is the full year in XXXX format. **“renderEvents”** is a boolean which defaults to false.

Operations

The function consists of a for loop which creates and appends the cells through JavaScript DOM elements. If **“renderEvents”** is set to true, it also creates a div with the ID of the current date to be rendered. Following this, it adds an event listener to the div, which when clicked, clears the sidebar and opens it to the create view with that date pre-applied to the start and end times.

renderEndPadding()

Objective

Renders the days of the next month that would be visible on the calendar for the current month.

Arguments

The renderEndPadding() function accepts 5 arguments: **cell**, **nextPaddingDays**, **month**, **year**, and **renderEvents**. **“cell”** is a reference to the div in which the cells should be rendered. **“nextPaddingDays”** is the number of cells to render. **“month”** is

the number of the month. “**year**” is the full year in XXXX format. “**renderEvents**” is a boolean which defaults to false.

Operations

The function consists of a for loop which creates and appends the cells through JavaScript DOM elements. If “**renderEvents**” is set to true, it also creates a div with the ID of the current date to be rendered. Following this, it adds an event listener to the div, which when clicked, clears the sidebar and opens it to the create view with that date pre-applied to the start and end times.

renderExtraPadding()

Objective

Renders the days of the next month that would be visible on the calendar for the current month (renders 7 more days than `renderEndPadding()`)

Arguments

The `renderExtraPadding()` function accepts 2 arguments: **cell** and **nextPaddingDays**. “**cell**” is a reference to the div in which the cells should be rendered. “**nextPaddingDays**” is the number of cells to render.

Operations

The function consists of a for loop, which creates and appends the cells through javascript DOM elements. The main purpose of the function is to simply render 7 more days than the `renderEndPadding()` function. This is exclusively used on the sidebar and the year page, where the calendars are too small to display events, but it is also desired to always have the month’s calendar consist of 6 rows.

clearCalendar()

Objective

Clears the calendar so that it can be loaded again.

Arguments

The `clearCalendar()` function accepts only 1 argument: **calendar**, which is a reference to the div which should be cleared.

Operations

The function sets the innerHTML attribute of “**calendar**” to an empty string, which deletes all children.

month.js

Global Variables

The month.js file contains four general variables: **const calendar**, **const urlParams**, **dateParam**, and **const dt**. The **calendar** variable is a reference to the calendar div. The **urlParams** variable uses the URLSearchParams() constructor to grab parameters from the URL. The **dateParam** variable searches the urlParams variable for a “date” parameter, and sets itself to that. The **dt** variable is set to a new Date object, using the default constructor which results in the current date. If anything other than an empty string is assigned to **dateParam**, then “dt” is updated to match that date; otherwise, the current value of “dt” is inserted into the URL as the date parameter.

load()

Objective

Loads the calendar for the selected month.

Arguments

The load() function accepts 2 arguments: **month** and **year**. “**month**” is meant to refer to a month; it defaults to “dt.getMonth()”. “**year**” is meant to refer to a year; it defaults to “dt.getFullYear()”.

Local Variables

The function contains a few local variables itself: **const firstDayOfMonth**, **const daysInMonth**, **const lastDayPrevMonth**, **const nextPaddingDays**, **const paddingDays**, and **eventData**. “**firstDayOfMonth**” represents the first day of the month, which is calculated by creating a new Date object from the month and year variables. “**daysInMonth**” represents the number of days in the month, which is calculated by creating a new Date object and passing “**year**”, “**month + 1**” and “0” into the constructor. “**lastDayPrevMonth**” represents the last day of the previous month, which is calculated by creating a new Date object and passing “**year**”, “**month**” and “0” into the constructor and calling the .getDay() method on that new Date object. “**nextPaddingDays**” refers to the amount of days from the next month that need to be rendered, which is calculated by subtracting a new Date object with the arguments “**year**”, “**month + 1**”, and “0” from 7. “**paddingDays**” represents the day of the week that the “**firstDayOfMonth**” variable falls on, which is calculated by using the .getDay() method on the aforementioned variable. “**eventData**” represents the full list of events for the user, which is received by calling the getEventsData() asynchronous function which is defined in calendar-common.js.

Operations

The “monthDisplay” element on the topbar of the UI is found and its innerText property is set to the month and year which is currently being rendered by converting **dt** to a string. A series of function calls are then executed:

The renderStartPadding() function defined in calendar-common.js is called with the arguments **calendar**, **paddingDays**, **lastDayPrevMonth**, **month - 1**, **year**, true. The renderMonth() function defined in calendar-common.js is called with the arguments **calendar**, **daysInMonth**, **dt**, **month**, **year**, true. The renderEndPadding() function defined in calendar-common.js is called with the arguments **calendar**, **nextPaddingDays**, **month + 1**, **year**, true.

The **eventData** variable is then iterated through to display any events which would be visible on the current calendar. In this for loop, a new Date object is created and is then formatted to be comparable to the IDs of the event container divs. If an event container div with the same ID exists, an event is rendered into that div; only three events can be displayed in the same container div. An event listener is then added to the event, so when it is clicked, it pulls up the edit sidebar with that event’s information and ID filled out in the form.

initButtons()

Objective

Allows navigation buttons to actually function and navigate the user to other month view calendars.

Local Variables

The initButtons() function contains two local variables, **nextButton** and **prevButton**. **nextButton** is a reference to the next button on the topbar of the UI. **prevButton** is a reference to the prev button on the topbar of the UI.

Operations

The operations performed by the function are simply adding event listeners to the buttons.

An event listener is added to **nextButton** so that when it is clicked, it calls clearCalendar(), adds one month to **dt**, replaces the URL parameters with the ID for the next month, and calls the load() function to reload the calendar with the new month.

An event listener is added to **prevButton** so that when it is clicked, it calls clearCalendar(), subtracts one month from **dt**, replaces the URL parameters with the ID for the previous month, and calls the load() function to reload the calendar with the new month.

year.js

Global Variables

The year.js file contains four general variables: **const calendar**, **const urlParams**, **dateParam**, and **const dt**. The **calendar** variable is a reference to the calendar div. The **urlParams** variable uses the URLSearchParams() constructor to grab parameters from the URL. The **dateParam** variable searches the urlParams variable for a “date” parameter, and sets itself to that. The **dt** variable is set to a new Date object, using the default constructor which results in the current date. If anything other than an empty string is assigned to **dateParam**, then “**dt**” is updated to match that date; otherwise, the current value of “**dt**” is inserted into the URL as the date parameter.

load()

Objective

Loads the calendars for the selected year.

Arguments

The load() function accepts 1 argument: **year**. “**year**” is meant to refer to a year; it defaults to “**dt.getFullYear()**”.

Operations

The “yearDisplay” element on the topbar of the UI is found and its innerText property is set to the year which is currently being rendered by converting **dt** to a string. A for loop is then initiated, which runs for 12 iterations and calls the miniCalendar() function in each iteration. The arguments passed to miniCalendar() are: a new Date object which copies **dt**, and i (the current iteration of the for loop).

miniCalendar()

Arguments

The load() function accepts 2 arguments: **dtRef** and **monthNum**. “**dtRef**” is a reference to a Date object. “**monthNum**” refers to the number of the month to render.

Local Variables

The function contains a few local variables itself: **const year**, **const firstDayOfMonth**, **const daysInMonth**, **const lastDayPrevMonth**, **const nextPaddingDays**, and **const paddingDays**.

“**year**” refers to the value obtained from **dtRef.getFullYear()**. “**firstDayOfMonth**” represents the first day of the month, which is calculated by creating a new Date object from the month and year variables. “**daysInMonth**” represents the number of days in

the month, which is calculated by creating a new Date object and passing “**year**”, “**month + 1**” and “0” into the constructor. “**lastDayPrevMonth**” represents the last day of the previous month, which is calculated by creating a new Date object and passing “**year**”, “**month**” and “0” into the constructor and calling the .getDay() method on that new Date object. “**nextPaddingDays**” refers to the amount of days from the next month that need to be rendered, which is calculated by subtracting a new Date object with the arguments “**year**”, “**month + 1**”, and “0” from 7. “**paddingDays**” represents the day of the week that the “**firstDayOfMonth**” variable falls on, which is calculated by using the .getDay() method on the aforementioned variable.

Operations

dtRef is first updated to match the current month which is desired to be displayed.

The “yearDisplay” element on the topbar of the UI is found and its innerText property is set to the month and year which is currently being rendered by converting **dt** to a string. A series of function calls are then executed:

The renderDaysOfWeek() function defined in calendar-common.js is called with the argument of the div to display them. The renderStartPadding() function defined in calendar-common.js is called with the arguments **monthCell**, **paddingDays**, **lastDayPrevMonth**. The renderMonth() function defined in calendar-common.js is called with the arguments **monthCell**, **daysInMonth**, **dtRef**, **monthNum**, **year**. If the child count of the parent div is less than 38, the renderExtraPadding() function defined in calendar-common.js is called with the arguments **monthCell**, **nextPaddingDays**. Otherwise, the renderEndPadding() function defined in calendar-common.js is called with the arguments **monthCell**, **nextPaddingDays**, **monthNum + 1**, **year**.

initButtons()

Objective

Allows navigation buttons to actually function and navigate the user to other year view calendars.

Local Variables

The initButtons() function contains two local variables, **nextButton** and **prevButton**. **nextButton** is a reference to the next button on the topbar of the UI. **prevButton** is a reference to the prev button on the topbar of the UI.

Operations

The operations performed by the function are simply adding event listeners to the buttons.

An event listener is added to **nextButton** so that when it is clicked, it calls `clearCalendar()`, adds one year to **dt**, replaces the URL parameters with the ID for the next year, and calls the `load()` function to reload the calendar with the new year.

An event listener is added to **prevButton** so that when it is clicked, it calls `clearCalendar()`, subtracts one year from **dt**, replaces the URL parameters with the ID for the previous year, and calls the `load()` function to reload the calendar with the new year.

schedule.js

Global Variables

The `schedule.js` file contains four general variables: **const calendar**, **const urlParams**, **dateParam**, and **const dt**. The **calendar** variable is a reference to the calendar div. The **urlParams** variable uses the `URLSearchParams()` constructor to grab parameters from the URL. The **dateParam** variable searches the `urlParams` variable for a “date” parameter, and sets itself to that. The **dt** variable is set to a new `Date` object, using the default constructor which results in the current date. If anything other than an empty string is assigned to **dateParam**, then “**dt**” is updated to match that date; otherwise, the current value of “**dt**” is inserted into the URL as the date parameter.

load()

Objective

Loads the schedule for the selected year.

Arguments

The `load()` function accepts 3 arguments: **year**. “**year**” is meant to refer to a year; it defaults to “**dt.getFullYear()**”.

Operations

The “**yearDisplay**” element on the topbar of the UI is found and its `innerText` property is set to the year which is currently being rendered by passing the **year** variable.

`getEventsData()`, which is defined in `calendar-common.js`, is called and the result is iterated through in a for loop. For each entry, if the currently viewed year lies somewhere between the start and end dates of the event, a new div is created and event information is filled out through creating and appending a series of JavaScript DOM elements.

When the loop is done, if the child count of **calendar** is equal to 0, a new element that says “No events scheduled for this year.” is created and appended to the page.

initButtons()

Objective

Allows navigation buttons to actually function and navigate the user to other schedules.

Local Variables

The `initButtons()` function contains two local variables, **nextButton** and **prevButton**. **nextButton** is a reference to the next button on the topbar of the UI. **prevButton** is a reference to the prev button on the topbar of the UI.

Operations

The operations performed by the function are simply adding event listeners to the buttons.

An event listener is added to **nextButton** so that when it is clicked, it calls `clearCalendar()`, adds one year to **dt**, replaces the URL parameters with the ID for the next year, and calls the `load()` function to reload the calendar with the new year.

An event listener is added to **prevButton** so that when it is clicked, it calls `clearCalendar()`, subtracts one year from **dt**, replaces the URL parameters with the ID for the previous year, and calls the `load()` function to reload the calendar with the new year.

week.js

Global Variables

The `week.js` file contains four general variables: **const calendar**, **const urlParams**, **dateParam**, and **const dt**. The **calendar** variable is a reference to the calendar div. The **urlParams** variable uses the `URLSearchParams()` constructor to grab parameters from the URL. The **dateParam** variable searches the `urlParams` variable for a “date” parameter, and sets itself to that. The **dt** variable is set to a new `Date` object, using the default constructor which results in the current date. If anything other than an empty string is assigned to **dateParam**, then “**dt**” is updated to match that date; otherwise, the current value of “**dt**” is inserted into the URL as the date parameter.

load()

Objective

Loads the schedule for the selected week.

Arguments

The `load()` function accepts 3 arguments: **day**, **month**, and **year**. “**Day**” is meant to refer to a day; it defaults to “`dt.getDate()`”. “**month**” is meant to refer to a month; it defaults to “`dt.getMonth()`”. “**year**” is meant to refer to a year; it defaults to “`dt.getFullYear()`”.

Operations

The “weekDisplay” element on the topbar of the UI is found and its `innerText` property is set to the first day, month, and year of the week which is currently being rendered by converting **dt** to a string.

`getEventsData()`, which is defined in `calendar-common.js`, is called and the result is iterated through in a for loop. For each entry, if the currently viewed week lies somewhere between the start and end dates of the event, a new div is created and event information is filled out through creating and appending a series of JavaScript DOM elements.

When the loop is done, if the child count of **calendar** is equal to 0, a new element that says “No events scheduled for this week.” is created and appended to the page.

`initButtons()`

Objective

Allows navigation buttons to actually function and navigate the user to other week view schedules.

Local Variables

The `initButtons()` function contains two local variables, **nextButton** and **prevButton**. **nextButton** is a reference to the next button on the topbar of the UI. **prevButton** is a reference to the prev button on the topbar of the UI.

Operations

The operations performed by the function are simply adding event listeners to the buttons.

An event listener is added to **nextButton** so that when it is clicked, it calls `clearCalendar()`, adds 7 days to **dt**, replaces the URL parameters with the ID for the next week, and calls the `load()` function to reload the calendar with the new week.

An event listener is added to **prevButton** so that when it is clicked, it calls `clearCalendar()`, subtracts 7 days from **dt**, replaces the URL parameters with the ID for the previous week, and calls the `load()` function to reload the calendar with the new week.

day.js

Global Variables

The day.js file contains four general variables: **const calendar**, **const urlParams**, **dateParam**, and **const dt**. The **calendar** variable is a reference to the calendar div. The **urlParams** variable uses the URLSearchParams() constructor to grab parameters from the URL. The **dateParam** variable searches the urlParams variable for a “date” parameter, and sets itself to that. The **dt** variable is set to a new Date object, using the default constructor which results in the current date. If anything other than an empty string is assigned to **dateParam**, then “**dt**” is updated to match that date; otherwise, the current value of “**dt**” is inserted into the URL as the date parameter.

load()

Objective

Loads the schedule for the selected day.

Arguments

The load() function accepts 3 arguments: **day**, **month**, and **year**. “**Day**” is meant to refer to a day; it defaults to “**dt.getDate()**”. “**month**” is meant to refer to a month; it defaults to “**dt.getMonth()**”. “**year**” is meant to refer to a year; it defaults to “**dt.getFullYear()**”.

Operations

The “dayDisplay” element on the topbar of the UI is found and its innerText property is set to the day, month, and year which is currently being rendered by converting **dt** to a string.

getEventsData(), which is defined in calendar-common.js, is called and the result is iterated through in a for loop. For each entry, if the currently viewed day lies somewhere between the start and end dates of the event, a new div is created and event information is filled out through creating and appending a series of JavaScript DOM elements.

When the loop is done, if the child count of **calendar** is equal to 0, a new element that says “No events scheduled for this day.” is created and appended to the page.

initButtons()

Objective

Allows navigation buttons to actually function and navigate the user to other day view schedules.

Local Variables

The `initButtons()` function contains two local variables, **nextButton** and **prevButton**. **nextButton** is a reference to the next button on the topbar of the UI. **prevButton** is a reference to the prev button on the topbar of the UI.

Operations

The operations performed by the function are simply adding event listeners to the buttons.

An event listener is added to **nextButton** so that when it is clicked, it calls `clearCalendar()`, adds one day to **dt**, replaces the URL parameters with the ID for the next day, and calls the `load()` function to reload the calendar with the new day.

An event listener is added to **prevButton** so that when it is clicked, it calls `clearCalendar()`, subtracts one day from **dt**, replaces the URL parameters with the ID for the previous day, and calls the `load()` function to reload the calendar with the new day.

sidebar.js

Global Variables

The `sidebar.js` file contains one global variable: **const main**. The **main** variable is a reference to the **sidebar** div.

initializeMain()

Objective

Renders the main sidebar view, which is just the create button and mini calendar.

Operations

Renders the create button and a mini calendar through creating and appending a series of JavaScript DOM elements. Calls the `initializeSidebarCalendar()` function to render the mini calendar.

initializeSidebarCalendar()

Objective

Renders the mini month-view calendar to the sidebar.

Arguments

The `load()` function accepts one argument: **calendar**. **calendar** refers to the element (div) to display the calendar in.

Local Variables

The function contains a few local variables itself: **const dt**, **const day**, **const month**, **const year**, **const firstDayOfMonth**, **const daysInMonth**, **const lastDayPrevMonth**, **const nextPaddingDays**, and **const paddingDays**.

“**day**” refers to the value obtained from **dt.getDate()**. “**month**” refers to the value obtained from **dt.getMonth()**. “**year**” refers to the value obtained from **dt.getFullYear()**. “**firstDayOfMonth**” represents the first day of the month, which is calculated by creating a new Date object from the month and year variables. “**daysInMonth**” represents the number of days in the month, which is calculated by creating a new Date object and passing “**year**”, “**month + 1**” and “0” into the constructor. “**lastDayPrevMonth**” represents the last day of the previous month, which is calculated by creating a new Date object and passing “**year**”, “**month**” and “0” into the constructor and calling the **.getDay()** method on that new Date object. “**nextPaddingDays**” refers to the amount of days from the next month that need to be rendered, which is calculated by subtracting a new Date object with the arguments “**year**”, “**month + 1**”, and “0” from 7. “**paddingDays**” represents the day of the week that the “**firstDayOfMonth**” variable falls on, which is calculated by using the **.getDay()** method on the aforementioned variable.

Operations

The function creates a date object and follows the same general structure as the month and year pages to render the month calendar.

The “monthDisplay” element on the topbar of the UI is found and its **innerText** property is set to the month and year which is currently being rendered by converting **dt** to a string. A series of function calls are then executed:

The **renderStartPadding()** function defined in **calendar-common.js** is called with the arguments **calendar**, **paddingDays**, **lastDayPrevMonth**. The **renderMonth()** function defined in **calendar-common.js** is called with the arguments **calendar**, **daysInMonth**, **dt**, **month**, **year**. The **renderExtraPadding()** function defined in **calendar-common.js** is called with the arguments **calendar**, **nextPaddingDays**.

clearSidebar()

Objective

Clears the sidebar to a blank state.

Operations

The function sets the **innerHTML** attribute of “**main**” to an empty string, which deletes all children.

initializeCreate()

Objective

Renders the “create” view of the sidebar.

Arguments

The function accepts one argument: **date**. **date** refers to a Date object, and defaults to the default constructor function, resulting in the user’s current time and date..

Operations

Through JavaScript DOM elements, creates a POST method form with the action “/api/events”. The form fields are: title, start time, end time, a reminder checkbox, location, and description. There are two buttons linked to the form: Back and Save. When the Back button is clicked, it calls the clearSidebar() and initializeMain() functions. When the Save button is clicked, it checks if the title field is empty; if it is, it gives it a generic title and submits the form.

initializeEdit()

Objective

Renders the event editing sidebar view.

Arguments

The function accepts one argument: **eventId**. **eventId** refers to the ID of the event which is desired to be displayed and edited.

Operations

The function searches through the list of all events tied to the user’s account, and when it is found, through JavaScript DOM elements, creates a POST method form with the action “/api/updateEvent”. The form fields are: title, start time, end time, a reminder checkbox, location, and description. All fields are non-editable. There are three buttons linked to the form: Back, Delete, and Edit. When the Back button is clicked, it calls the clearSidebar() and initializeMain() functions. When the Delete button is clicked, it changes the form action to “/deleteEvent” and submits the form, resulting in the event being deleted from the database. When the Edit button is clicked, it makes all fields editable, removes the Delete button, and replaces the Edit button with a Save button. The Save button will save and overwrite the event in the database.

topbar.js

Global Variables

The topbar.js file contains two global variables: **const dropdown** and **const dropdownContent**. The **dropdown** variable is a reference to the **dropdown** div. The **dropdownContent** variable is a reference to the **dropdown-content** div.

initializeDropdown()

Objective

Creates the dropdown menu on the right side of the topbar.

Operations

Adds an event listener to the **dropdown** so that when clicked, it toggles the dropdown menu. Adds an event listener to the window, so that when the user clicks anywhere outside of it, it closes the dropdown menu. Creates and appends links to the other pages through a series of JavaScript DOM elements.

login.js

Global Variables

The login.js file contains only one global variable: **const main**. The “**main**” variable is a reference to the **main** div.

initializeMain()

Objective

Renders the word Calendar+.

Operations

Creates and appends to “**main**” an h1 with the text “Calendar+” through JavaScript DOM elements.

initializeLogin()

Objective

Renders the login page / form.

Operations

Through JavaScript DOM elements, creates a POST method form with the action “/api/login”. The form fields are: username and password. There are two buttons linked

to the form: Sign Up and Log In. When the Sign Up button is clicked, it calls the `clearMain()`, `initializeMain()` and `initializeRegistration()` functions. When the Log In button is clicked, it checks if all fields in the form are valid, and if so, submits the form.

`initializeRegistration()`

Objective

Renders the registration page / form.

Operations

Through JavaScript DOM elements, creates a POST method with the action “/api/user”. The form fields are: username, email, password and confirm password. There are two buttons linked to the form: Back and Sign Up. When the Back button is clicked, it calls the `clearMain()`, `initializeMain()` and `initializeLogin()` functions. When the Sign Up button is clicked, it calls the `checkPasswordsMatch()` function, checks if all fields in the form are valid, and if so, submits the form.

`clearMain()`

Objective

Clears the login/registration page so that it can render a different view.

Operations

The function sets the `innerHTML` attribute of “**main**” to an empty string, which deletes all children.

`checkPasswordsMatch()`

Objective

Checks if the passwords on the registration form match one another.

Operations

If the passwords on the registration form don’t match one another, the function uses the `.setCustomValidity()` method to notify the user that the passwords don’t match.

`index.js`

Objective

Handles the entire application tier of the Calendar+ app.

What does it do?

This file first connects to the database, printing an error message to console if there is an error. It uses server-side cookies to keep track of user credentials so that the correct CRUD operations can be executed. Ultimately, it is responsible for handling the request-response cycle, CRUD operations, and notifications. All routes used by the website are handled in the index.js file, meaning that all GET and POST requests are handled in this one file. The server connects to the database using a .env file to store the database username, password, and database name.

There is a route for each page so that the server does not redirect to the static .html pages and instead uses the route to send the file. This approach keeps users from being able to access pages while not logged in.

The “/EventsData” route sends a json array of events queried from the database based on that user’s username gathered from the express-sessions cookie they are assigned when they log in.

The “/api/user” route inserts the data retrieved from the sign up form into users table within the database.

The “/api/login” route queries the database for the user information within the login form and if no user exists with that data, then they are redirected back to the login page, otherwise they are redirected to the month page.

The “/updateEvent” route updates the event at the id retrieved from the form that the user edits the event data from.

The “deleteEvent” route Deletes an event from the events table within the database that matches the event_id given.

The “/api/logout” route destroys the express-session value within the server so that the user must login again to utilize the website.

System Architecture

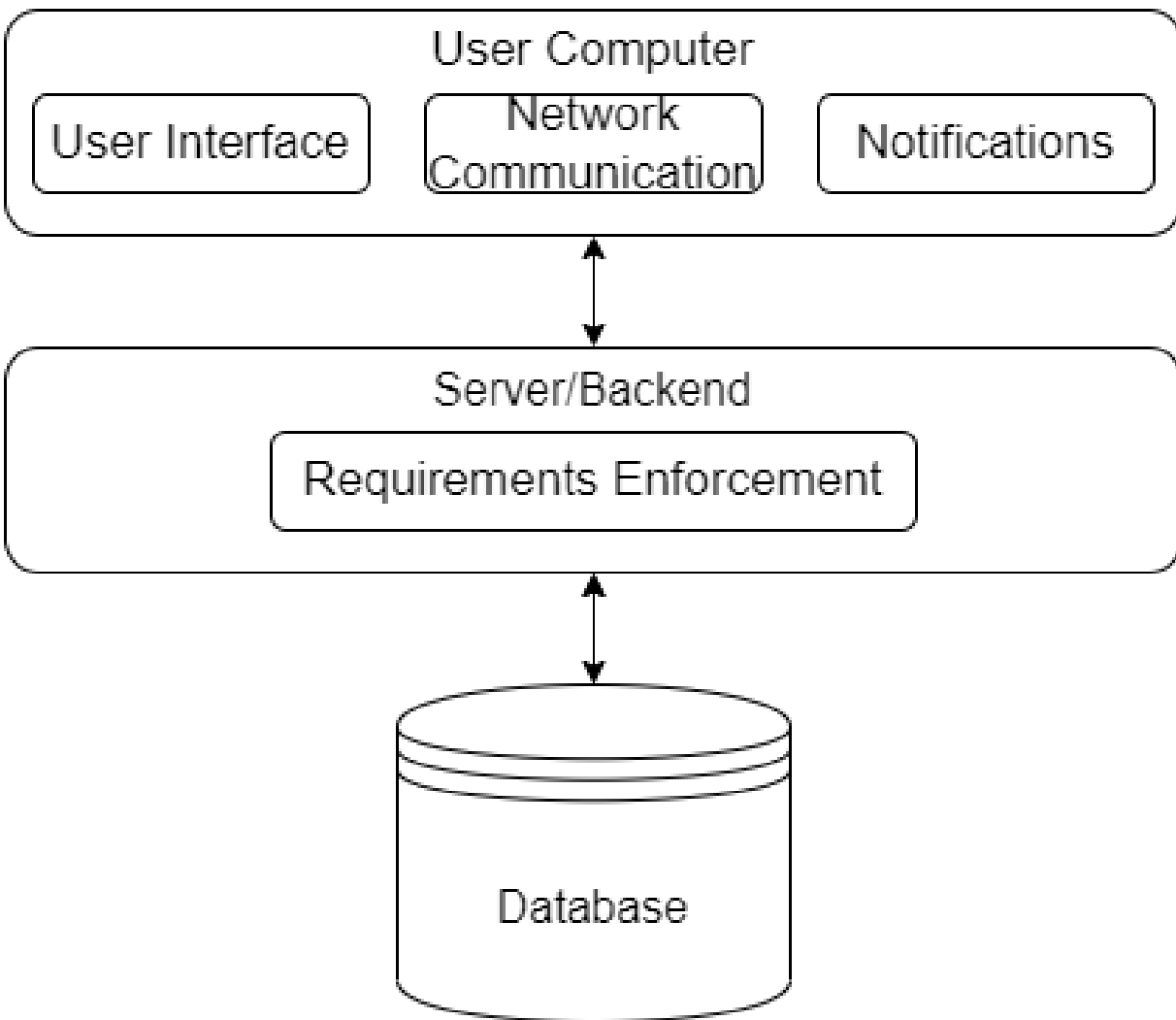


Figure 12 - Block Diagram of System Architecture

The block diagram of the Calendar+ web application illustrates a three layer system composed of the user computer/browser, the hosting server, and the database. The user computer is the presentation layer of the Calendar+ web app; it represents the visible parts of the web application to the user and is responsible for network communications with the backend as well as providing system notifications to the user so that reminders can be seen by the user. If the user enacts an action that does not require any response from the backend, such as switching to the yearly calendar view, the browser layer will complete the action itself without having to communicate with any other layer.

Whenever the user desires a task involving the backend such as account login to be completed, the user computer sends a request to the server hosting the Calendar+ application. The server is responsible for validating the user-defined task. In the event that the user request contains some sort of error, the server will send the request back

to the user computer which is then responsible for showing the error on the user's screen. For example, if the user tries to login with incorrect credentials, the server will deny the login attempt and communicate to the user computer to display an error to the user. In any case, the server is ultimately responsible for enforcing the restrictions of the Calendar+ web application. It serves as an intermediary between the user computer/browser layer and the database, reading data from the database and transferring it to the browser layer and writing data to the database as requested by the front-end as necessary.

The database is by far the simplest part of the system. It is responsible for holding all application data across all users. As all verification and validation is completed by the server layer which is the only layer the database communicates with, the database itself does not need to worry about potential errors in its data.

Overall, the benefits of this closed three-layer system are clear. It offers simplicity, clear separation of concerns, and a degree of modularity that makes it easy to change out individual layers or to add additional layers if extra functionality is required. However, the drawbacks are also clear and enormous. Because this system is a single unit, making a change requires redeploying the entire system. Because data access requires a chain of communication that goes from the browser layer all the way down to the database and then all the way back up to the browser layer, this layered architecture has no scalability. Its modularity is limited to the individual layers. And lastly, if one layer fails, the entire system also fails. There is no fault tolerance.

Database

For the database portion of Calendar+ we have used MySQL. If you refer to Figure 13 below, our database, at current, features two tables: Events, and Users. The Users table features three attributes: user_name, Password, and Email. The user_name attribute is the primary key meaning that it is a unique value and shall be used to access data from the Events table. The user_name attribute is of the varchar data type and will at maximum be 250 characters long and must have an entry within the table. The Password attribute is of the varchar data type and will at maximum be 250 characters long and also must have an entry within the table. The Email attribute will also be of the varchar data type and will at maximum be 250 characters long and must have an entry within the table. All of the entries of the Users table are inputted during account creation.

The second table named Events will feature nine attributes: event_id, user_name, event_title, eventDate, startTime, endTime, eventLocation, eventDescription, reminderDate, eventColor. The event_id attribute is of the integer data

type and shall auto increment, event_id is the primary key of the Events table. The user_name attribute is a foreign key relating the Events table to the Users table and must have an entry within both tables. The user_name attribute will relate to all events created by that user_name, thus allowing us to query the database on events created solely by that user's account to populate the calendar with their events. The eventDate attribute is of the Date data type and will store the date of each event in the YYYY-MM-DD format; the eventDate is required for all entries of the Event table. The startTime attribute is of the Time data type, will store the beginning time of each event in the hh:mm:ss format, and is required. The endTime attribute is of the Time datatype and will store the end time of each event in the hh:mm:ss format and must be greater than the startTime of this same event. The eventLocation attribute is of the varchar data type and will at maximum be 250 characters, but can also be null. The reminderDate attribute is of the Date data type and is optional and will store the reminder date of each event in the YYYY-MM-DD format if the user wants a reminder. The reminderTime attribute is of the time data type in the YYYY-MM-DD format and is required if the user enters a reminderDate but will not be required if the user does not enter a reminderDate. The reminderDate attribute is checked and if this attribute is not null then the reminder system will alert the user at the chosen time and date. The eventColor attribute is of the varchar attribute and is 6 characters long. The eventColor attribute is stored with hexadecimal data for the color the user wishes the event to appear on the calendar, but is currently unused. All of the entries of the Events table are inputted during event creation.

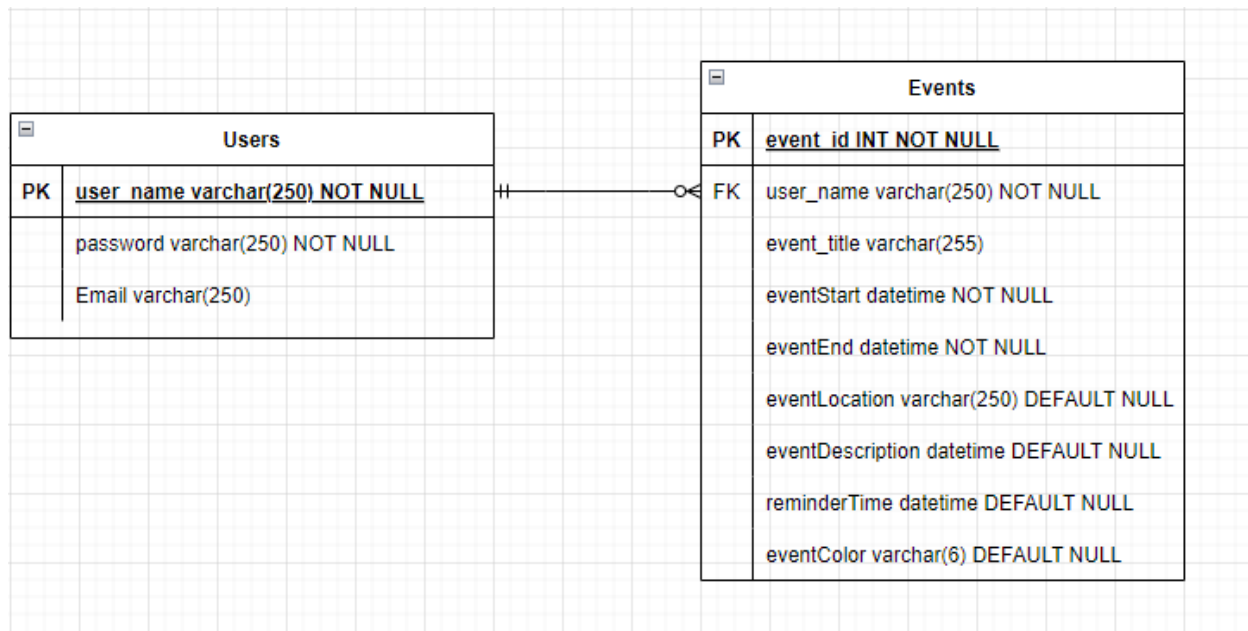


Figure 13 - ERD diagram of the server database

Known Bugs

Using quotation marks inside any fields during event creation will cause the entire website to crash. On the year view, every 11 or so years, February will appear with only 5 rows of dates instead of 6 like every other month. If an event spans multiple days, because of the way the schedule, day, and week views are set up, it will appear on the page for every one of those days but it will only show the start date and times for every single one.

Future Work

There are many areas for future improvement of Calendar+. The first and most important feature we need to add is the ability to be reminded of events. While it was originally in scope to be released in version 1.0, some issues arrived with the module we were using. Another future improvement is the implementation of multiple calendars, which would allow users to toggle on/off a set of events. Such an improvement would allow users to better organize their events.

We would like to implement unique views for the day and week views, rather than having them piggyback off of the code which was written for the schedule view. This would likely see them transformed into something more akin to Google Calendar's day and week views, where each event takes up a proportional width and height of the screen over top an underlying grid which lists all hours of the day.

Another area for future work is to implement an advanced search feature, allowing users to find specific events in their busy schedules without having to search through many pages to find the one they want. It would also be natural to implement Google Maps integration for those events, allowing users to obtain directions to destinations. To-do lists and tasks implementation would also add considerable utility to Calendar+.

We would also like to create a mobile port of the application, so that it is accessible on more than just desktop. Creating a mobile port will make Calendar+ dramatically more accessible, as mobile devices make up roughly 68% of all web usage.