Orlando Math Circle Design

An event calendar system with a native app-like experience in the browser.

Orlando Math Circle

Math & Creativity | Service & diversity

Group 10

John Gilbert
Mark Howell
Jason Mathew
Courtney Stewart
Joseph Tolber

Sponsors

Sheina Rodriguez
Matthew Villegas

Table of Contents

1. Executive Summary

In a world where STEM is a rapidly expanding industry, it is more important than ever to cultivate and encourage the next generation's interest in mathematics. Mathematics is an ubiquitous tool that promotes problem-solving and critical thinking skills important to even those who are not interested in STEM careers. Orlando Math Circle seeks to accomplish this and more, especially for disadvantaged minorities and girls, by pairing up educators, volunteers, and children through their nonprofit organization. We are here to provide OMC a mobile calendar event system and a contact portal to replace their currently nonfunctional native application.

The calendar shall feature individual events that parents can sign their children up for, volunteers can select jobs and accrue volunteer hours, and an administrative system that allows for events to be created where they can limit who can attend based on age, gender, or by limiting the total number of registrations. These requirements will be met with a customizable role system that allows administrators to modify the permissions of individual users and a project system that events inherit from allowing volunteers to earn volunteer hours based on tasks performed for a project.

Our clients expressed two main concerns outside of the operational requirements of the website. First, the previous application was closed-source and wherever possible the new website should be able to be cloned and interacted with as a learning tool for select students of theirs. Secondly, when the application did work registration took numerous steps and the demographic of parents registering their children have a low technological aptitude and often utilize outdated mobile hardware. This poses unique considerations as COPPA compliance requires we do not allow children to enter any personally identifiable information within our application without an extensive parental approval process, and the information we store will require careful consideration for security, often more difficult if you involve unknown users in the future development process. When it comes to the ease of registration and account usability, one of the parallels we are looking to draw from was the Netflix account scheme. It allows for multiple sub-accounts within a primary account which would allow for robust user permissions while requiring less effort on part of the parents.

2. Project Overview

2.1 Project Description

Orlando Math Circle is a non-profit organization focused on enabling students to pursue mathematics outside of school. They are determined to help students of all social and ethnic backgrounds, but also wish to focus primarily on those who may not have access to many math programs outside of their school. The objective of this project is to create a new website for Orlando Math Circle, and to make it a website that is easy to access and use for all users. The clients' list of requirements for the project was formulated with this in mind.

First, the clients wish for the registration process to be as streamlined as possible. The current Orlando Math Circle website has a fairly user-unfriendly approach to registering for an account, especially when users need to add their children to their account, and the new website will need to remedy this. The registration process also needs to allow for different account types, meaning parent, student, volunteer, etc. Second, the website will need an event calendar that administrators will be able to add events to. Orlando Math Circle operates through events where they tutor students in mathematics, with some of these events being one-offs, and some being regular occurrences. Third, parents need the ability to submit forms and pay fees through their account. There are various forms and fees that a parent will need to submit or pay, and the current website does not offer an intuitive way to do this. Lastly, Orlando Math Circle administrators will need to have special accounts with a number of special privileges. The current website does not allow administrator accounts to change many things they need to change, and the clients have specifically requested that we add these privileges for them. These requirements, and others that will be detailed elsewhere, are what the clients wish to receive from their new website.

2.2 Goals & Objectives

The current native app is closed source, has few features, is difficult to use, and requires too many steps in the registration process. The goal of this project is to create a website with a native mobile-application feel that will improve the usability of the previous app, increase the turnout to the OMC events, and create an incentive for participation. There are 4 main objectives we need to reach to get to our main goal of this project. These

objectives are to improve usability, provide a payment system, create an event calendar, and implement a loyalty system.

We plan to improve the usability by simplifying the account signup process and differentiating account types. Currently, the account sign up process is seven steps and cannot be done from a mobile browser. We need to create an account layout that is easy for parents to use and gives children some abilities while also restricting their power to access or change certain elements of the account. To do this we are planning on using a Netflix type of login process. There will be one login username and password for each family, and when you sign in the first thing that will pop up will be the user selection. This will give children the access they need to their events, points, questions, and any other parts that do not need direct parent involvement. The parents' user page will have an additional password or PIN that will need to be put in before allowing access to the more sensitive information. The main parts we will need to keep protected will be any user information -- names, birthdays, emails -- as well as payment information and documents. This setup will also keep us in compliance with COPPA as the children will not be giving any of their own personal information. This was one of the most important design challenges for us to get right so that it allows for an easy flow for parents, as much freedom as possible for the children, all of the information needed for OMC, COPPA compliance for collecting user data, and an achievable process for us to build.

We also plan to provide a payment system. To do this we are going to integrate PayPal into the website. The main use of this payment system is to allow parents to pay for any event costs upfront when signing up their children. OMC already has a PayPal account so we will just need to work with them to get their account information and add the feature to the website to make the process more seamless. This will give OMC the capacity to make payment for events ahead of time possible and therefore making registration even easier for parents. Overall, it will help move them towards the goal of an easier and more user-friendly website setup.

To increase turnout, we will create an easily editable event calendar system that the admins can use to post and update events, as well as put restrictions for age groups or other variables. This is the other really big piece of the website aside from account setup because most pieces of the product are based around this center idea of a calendar system. The plan is to have the calendar system display events and allow participants to see and sign up for events online, also easing the registration process. There will also be a hierarchy of projects and events, where projects will be larger tasks that can have

multiple smaller and focused events branched off of it that work towards the completion of the project. Events on the calendar will show whether it is a part of a bigger project or if it is its own stand alone event. This will also give volunteers the ability to choose to work on events that are part of the same project. Building a clear calendar at the center of this website gives OMC the chance to increase student turnout by providing them with an efficient online signup process, as well as allowing them to set up their events better with more information about the students they'll have attending beforehand.

The loyalty system is meant to incentivize the participation of students and volunteers. This is going to be done through a loyalty point system for students based on attendance and for volunteers based on volunteer hours. We will have a section of each user profile where the total points will be calculated and shown. The idea is to have a certain number of points assigned to each event for students attending and a different number of points for each job that a volunteer can sign up for at the events. These points can then be used to encourage signups for less popular events or more demanding volunteer jobs possibly with larger loyalty point amounts.

2.3 Broader Impacts

The broader impact of this website for the Orlando Math Circle will have more of a local impact on society. A large part of what the Orlando Math Circle does is working to engage K-12 students in STEM areas through different events such as math competitions or computer science skills weeks. Therefore, creating this website for them gives them the opportunity to get attendance and engagement in these events to increase by giving them a platform to actively communicate with students and parents. Their biggest problem currently is that their current website is not user-friendly, nor developer-friendly, and so a lot of their participants are missing out on information or events due to lack of communication. They have said themselves that solving this problem will lead to increased participation and give them a wider outreach to bring in and educate more students.

Overall, this project will impact underrepresented groups in the local area as their main focus is bringing these classes and events to disadvantaged students. One of the goals they want is to have a place for students to develop their computer science skills in web development and be able to add their work to the website we build. Creating the ability for them to have this will give these students skills that they can use to help them in school and even in their communities. Orlando Math Circle is a nonprofit organization so

having the ability to get this website developed by our senior design team gives them an opportunity to revamp their system that they probably would not have had the budget to pay a developer to improve.

2.4 Statement of Motivation

2.4.1 John

Full-stack website development has been of interest to me since my early years of high school. My school had a site that used some drag-and-drop software that was pre-Dreamweaver and I was called in to fix it occasionally. Yet one day it broke in a rather extraordinary way and I was able to help transition them to a more modern website and negotiated with the school district's webmaster to get the best solution for our school.

When you work on projects yourself you are your own boss with your own vision and things do not get lost in translation. You may not be able to build the best website because when you are building it by yourself the person you are looking to impress can only ever be yourself. I hope to learn a lot from having clients and being able to get to the root of a problem and pick the best solution for it rather than what I am comfortable with. Not only that, being able to make a product that makes someone happy is going to be quite fulfilling.

2.4.2 Mark

I have been interested in website development since high school, and this project offers a unique opportunity to learn more while also helping a good cause. I have volunteered to help tutor kids in programs like Junior Knights, and I have volunteered to help run the UCF Annual Coding Competition, but I have never had the chance to really help a tutoring organization as much as this project will. This project will not only allow me and my teammates to learn more about web development, helping us find jobs after college, but it will also be a tremendous help to Orlando Math Circle in helping more kids get tutoring in math. I am looking forward to working with my team on this project and helping our sponsors to improve their website.

2.4.3 Jason

In high school, I was an officer for a math honor society known as Mu Alpha Theta. Along with club administration decisions, I was able to tutor students in computer science and various math classes. This project stood out to me because of its involvement in

helping students and volunteers. I have basic front-end experience with Android and Qt, but I have always been interested in learning new things. This is why I chose a project that would allow me the opportunity to learn web development as well as work on the back-end. I hope to work well with my team and learn about working with sponsors.

2.4.4 Courtney

I am personally interested in this project because I have always had an interest in math, I started at UCF with a math minor and I was in the math honor society in my high school. I also really appreciate what the organization is doing with bringing these skills and opportunities to children who might not have had the possibility to be involved in STEM early on without this program. I have volunteered for STEM Day at UCF a few times and always appreciated seeing how much the kids enjoyed the day. Hopefully, we can make the Orlando Math Circle a website that gives them a better way to connect with people and help them in their goal to engage more children in STEM.

2.4.5 Joseph

Ever since high school, I have had an interest in math. I took AP Calc 1 and 2 in my Junior year and TA'd in the Pre-Calc and Stats classes. I also would help my Calc teacher tutor students on some weekends at the school. I picked this project because of how it involved volunteering and helping kids get involved with math. I have some experience doing back-end development such as creating databases and APIs. Additionally, I would like to use this project to get an understanding of front-end technologies and languages so I am looking forward to getting started with my team.

2.5 Legal, Ethical, & Privacy Issues

2.5.1 COPPA Compliance

Our website will be collecting information from children under 13 once it is implemented, as many of the children tutored by OMC fall into this age bracket. As such, we will be required to follow the guidelines set forth by the Children's Online Privacy Protection Act (COPPA). The first of which is to construct a privacy policy to notify users of several key factors:

- A list of all parties that will have access to the information gathered
    - This includes the website and any 3rd parties, including advertisers (if any are present)

    - Contact information will be listed for all parties
- A description of what information will be gathered and how it will be gathered
    - Indicate the exact information that users will be asked to provide (name, email, address, etc.)
    - Indicate how it will be gathered, either by manual input or through the use of cookies
    - How the information will be used
    - Whether or not it will be disclosed to other parties
- A description of parent rights
    - No more information will be gathered than is necessary
    - They reserve the right to edit or delete any information given
    - They reserve the right to stop any and all collection of their child's information at any time
    - They can decide not to allow their child's information to be disclosed to third parties, while still allowing the information to be collected by the site

This privacy policy must be sent to parents directly and it must be clear and easy to read. In addition to this privacy policy, parents will need to provide consent if their children's information is collected directly from the child, and will need to be notified of any changes in our privacy policy should any occur. The parents will need to provide consent through one of several ways, among which are a signed consent form, submission of a government-issued ID (to be deleted after), or verification using a photo ID. Once consent is provided and the child's information is gathered, parental rights will need to be honored at all times. This means that we must provide a way so that, should a parent request it, they can review all information collected about their child, delete this information and revoke consent and refuse the further use or collection of personal information about their child.

Lastly, all necessary steps must be taken to ensure the gathered information of all children on the website is kept secure. We must take reasonable steps to only release information to providers and third parties who will be capable of maintaining its security, confidentiality, and integrity. We should also only hold on to this information as long as it is necessary and securely dispose of it once it is no longer needed.

2.5.2 User Data

The application will also collect the names, ages, emails, and other data from the parents using it. While this is not a legal issue like it is with the children, it is information that the

users will definitely want to be kept private within the application. We will have to encrypt the passwords they use at a minimum and any other documents or data that needs to be kept secure.

SSL encryption will be required for all levels of networking communication within the website that are not on local hardware. The hardware itself should not be accessible to students. Only the repository code should be given to students. API keys are highly sensitive pieces of information that should not be exposed to those not tasked with managing the server. Similarly, the application will be coded with a configuration system to protect sensitive information as it cannot be hardcoded into an open-source application. Development should not be done on the production database or server and instead locally as errors in development may leak sensitive information.

2.5.3 PayPal

The clients have specified that users should be able to use PayPal to pay for any fees that may be required for an event. Credit cards and other secure user-identifying information may, and likely will, be entered on the website if the PayPal Smart Buttons are used by entering payment information without immediately redirecting to PayPal. We will be storing transaction details that tie users to having made payments, but payment credentials and user addresses will not be stored within the website's databases. Nonetheless, payments will be made on the website and the credentials as well as the transaction flow should need to be protected from man-in-the-middle attacks, or where someone is able to intercept and record or potentially modify requests by placing themselves between both parties. PayPal has minimum requirements for SSL encryption that will be followed, otherwise the service will reject requests from the website.

3. Specifications & Requirements

The function of the project is to replace the current native app with an app-like website. This website cannot be a progressive website application (PWA) but must have a similar look and feel to a native application. The primary goal is for the website to function as a robust event calendar system that is flexible for administrative users to edit.

The clients have an express desire that the application prioritizes ease of use for each age group above most other requirements. However, many of the functional considerations will also need to keep in mind that COPPA restrictions are in full effect and we will not be able to allow children under the age of 13 to input any identifiable information and methods of communication will be limited. Secondary goals focus on the administrative management of user accounts, emailing aggregation, and volunteer point systems. Tertiary goals focus on design or engagement features that are meant to improve engagement with the children or entice them to use the service.

3.1 Operational Requirements

1. The application shall be able to handle 70 concurrent users.
2. The application shall feature a calendar event system that allows the creation and editing of events and the ability for users to register to these events.
3. The application shall allow admins to view who is going for each event.
4. The application shall have separate permissions for admins, parents, and students.
5. The application shall have the ability for users to be marked as volunteers or educators.
6. The application shall allow admins to limit the number of event registrations.
7. (Homerun) The application shall send email notification 2 weeks before, 1 day before, and 1 hour before an event.
8. (Homerun) The application shall send emails for all event modifications to attending users.
9. The application shall allow for projects to be created with a description and optional picture that events can optionally inherit.
10. The application shall allow for projects to have pre-defined volunteer jobs.
11. The application shall allow admins to pull up a roster parents utilize at events to mark themselves present.
12. The application shall require students under 13 to have parents create accounts for them.

13. The application shall allow events to be customizable, limiting registration to certain age groups, genders, or grade ranges.
14. The application shall use PayPal to allow parents to pay for event costs.
15. The application shall allow admins to add overrides to accounts with "free/reduced lunch" statuses and bypass payment.
16. The application shall allow admins to create and edit other accounts.
17. The application shall require volunteers to affirm they have read a *Volunteer Handbook Agreement* document.
18. (Stretch) The application shall port over any attainable account data from the old system.
19. (Stretch) Parents should be able to upload files for admin verification (pdf, jpg, png, or heic file types).
20. (Stretch) The application shall display social media feeds from Facebook or Twitter OMC posts.
21. (Homerun) The application shall utilize a loyalty point system awarding users with points for attending events and allowing them to gain levels associated with mathematicians.
22. (Homerun) The application shall assign volunteer hours for those volunteering for events.
23. (Homerun) The application shall include a framework for multiple-choice math challenge questions to be created and answered to gain points.

3.2 Design Requirements

1. The application homepage shall display upcoming events and those signed up.
2. The application backend shall display all users and allow them to be filtered.
3. The application backend shall allow for emails to be sent to all users matching account criteria.

3.3 Documentation Requirements

1. (Homerun) The application shall include developer documentation so the application may be modified or used for educational purposes in the future.

4. Division of Labor

We have decided to work on the website using tiered phases where different members focus on a specialized responsibility depending on the phase of development we are currently in.

- Phase 1: Entity modeling and management. Every member of the team will focus on the backend logic for each entity..
- Phase 2: Prototyping and Entity Systems. The team will split into two groups to tackle the first layer of connectivity between the frontend and the backend. The first group works closely with Sheina to add functionality and basic design to our prototypes. Simultaneously, the backend group will be implementing the flow for each action the user will be taking on the website. Some processes may still be mocks, such as external APIs.
- Phase 3: Data Flow. The team pivots as a whole to applying the logic to connect the frontend to the backend layers. At the end of this phase, the prototype of the website will be able to complete the basic requirements expected from the clients. Rough evaluations will be had with the clients.
- Phase 3: Integrations and Design. Proper design will be added to the website in this phase with a focus on usability and testing the different responsive views on various devices. A smaller backend team will work on replacing previously temporary features such as PayPal integration and emailing with their sandboxed alternatives.
- Phase 4: Evaluations and Administrative Tasks. The team will be at a place where it can have multiple rounds of evaluations and make adjustments to the website as necessary. Things like SES, PayPal, the VM, and the domain will need to be implemented early into this step to ensure proper compatibility. Stretch goals, if not previously already implemented, will be reviewed in this stage if they will make the final version.
- Phase 5: Delivery. Documentation will be concluded from each member on how to utilize the website as an administrator and how to approach it as a developer. How to configure, build, and run the website will be a forefront staple.

4.1 John

During the backend-oriented phases of development, John will be managing the resources, hardware, accounts, and repositories for the team. Controller routing, input validation, authentication, configuration, and hardware software such as NGINX, Node.js, and PM2 will be handled as well.

When the current goals are frontend development, John will be responsible for the authentication and authorization layer, storing credentials and other state management requirements, form validation, configuring the frontend libraries for handling requests during server-side rendering vs. SPA mode, and describing error management.

4.2 Mark

The calendar will be Mark's primary focus as the backend is being constructed. The necessary tooling and logic for creating singular events or events that require recurrence rules and how to describe them will also be in his domain. Each event entity and its associated entities and end-to-end tests will be designed by him as well.

Mark will work closely with the clients during the prototyping stages of the frontend as a liaison to relay the current designs and collect feedback from them or other OMC-related personnel. He will also be adding the initial data structures to prototypes and developing the component outlines for iteration by the team.

4.3 Jason

PayPal, AWS SES, the Redis-backed queue, and other external services will be managed by Jason during the development of the backend. His job is to scope out the requirements of the clients and determine how to best match the external services and how to incorporate them.

As the initial frontend work is started, Jason will work closely with Sheina to make mockups of her design ideas for the website. Mark will be consulted to translate the designs into data-less components that Mark will then iterate on.

4.4 Courtney

As the backend is being developed Courtney's primary job will be to describe the entity schemas in MikroORM, setup the PostgreSQL database, and create the initial migrations. A strategy for backing up the database, handling updates to production data, and any attempts at past data migrations will also be handled by her.

Once the frontend is at the data integration stages, Courtney will rectify the component prototypes with the entity schemas and setup the Vuex modules and ensure each backend controller is returning the necessary data.

4.5 Joseph

The abstract entities and processes that require approval will be handled by Joseph. He will be responsible for the file uploading, file storage, system security, and the project and volunteering points system processes. Joseph will also be undertaking the attendance data.

Once the frontend is in its data states, Joseph will be working to implement the pub-sub or websocket systems necessary for the attendance pages and handling the logic for pendable items, such as volunteer hours approval, and free lunch forms.

5. Research

5.1 Potential Solutions

These are the initial thoughts of the team on the various requirements and features described by the clients. These ideas may have been viable or they may not have, but they are the surface-level ideas for how the system could be designed prior to research.

5.1.1 Registration Solutions

1. Following the client's initial plan, during registration we can confirm if the registering user is 18 or older. If they are, they can then make an adult account that has the ability to create child accounts either with the same email and different password as the adult account, or they can input child emails or usernames and passwords. This has a high disparity from most website registration systems, and is less usable for high schoolers. Children also have little reason to have accounts. In a similar vein, this solution may have concerning ramifications for children who lie about their age, a noted issue with the previous solution by the clients.
2. Registration can be done at 13 years or older, or with parent permission, and the accounts are familial like a Netflix system. The account holder specifies the users within the account and includes associated names, dates of birth, and emails as necessary or desired. Other defined users within an account will have separated permissions and website experiences coupled tightly through one account.
3. Minors 13 and older can register for their own accounts with parent permission, obtainable either by electronic signature or submission of a signed form. Once a minor has created an account, it will be linked with another parent account. If this minor account attempts to register for any events, an email will be sent to the connected parent account to ensure the parent has consented to their child attending said event. This is also not very user-friendly in that it requires extra steps for parents to link their accounts to their children. This approach would be closely following COPPA suggestions, but it infringes upon the client requirement that the registration is quick and easy.
4. Child information will require dynamic permissions based on their age and grade level. The application will need to be able to securely determine the age of each user and allow them to access the features of the website without displaying it publicly. Similarly, age has a loose association with grade level and parents are

unlikely to routinely update this information. It is uncertain if we should guess the grade of a student based on their age, or if the parent enters this information once. If parents only enter this information once, the value they enter would become invalid in the next school year, requiring that every child's grade level be calculated relative to how many school years it has been since this value was updated.

5. Volunteers can be anyone who is at least 13 years old. With the client's original plan, volunteers would have their own account type separate from the parent or child accounts. This creates an issue where children or parents are not able to volunteer for events without a separate volunteer account. Instead, we could simply have adult accounts for parents and adult volunteers. Since children are also able to volunteer for events, those accounts could potentially be flagged as volunteers.

5.1.2 Calendar Solutions

1. Codependence on Google Calendar APIs could allow for easier setup of the calendar system without having to recreate complex timing logic. This may lower performance as each event modification has to not only save in the database, but also with Google. It would not be simple to allow modifications of the calendar directly from Google's end as we would have to poll the API, listen to webhooks, or other Google events to mirror the changes. This does not rule out the need for a looping action on the server to check when notifications need to be sent.
2. Alternatively, we can use an event table and an event loop that is not associated with a Google Calendar. This will limit complexity but may make recurring events that much more complicated if we wanted to preserve metadata across them.
3. Emailing users about events at scheduled times will require a queuing system, persistent loop, or cron tasks.
4. The calendar could utilize push notifications to notify users of upcoming events. They would be sent at certain periods of time prior to the event to ensure the user is aware of when and where the event is taking place. It could be done, however the clients are concerned with popups asking for installations or notifications being either intrusive or confusing to the demographic of parents.

5.1.3 Student Integration Solutions

1. One of the client's stretch goals is to have space for students to learn computer science skills with their website. Ideally, they want students with the proper knowledge to be able to perform maintenance on the website after we are done with it, and without us dumbing down how we choose to implement it on our side. We are already allowing admins to assign roles to different accounts as a must-have requirement, so we could have an option where they give certain student accounts the ability to update the website. This, of course, opens them up to a lot of possible mistakes and breaking of the website.
2. Another option is to allow them to clone the GitHub used in the development of the website so that they can see how certain changes are applied and it allows them to work on their skills without risking the functionality of the website. This allows the admins to have multiple students work on fixes and other maintenance they need without an immediate effect on the website. They can then take any solutions that worked and implement them if needed.
3. If the clients are more focused on having a space to allow students to work on web development and less on having them be the ones to upkeep the website, we could simply have a separate page on the website for student work. Each student could have a link off there to a page that they created where they could work on front end development. This gives them no access to any other parts of the website and therefore no risk to breaking any other parts of the website.

5.1.4 Event Registration & Attendance Solutions

1. Event registration will have dynamic options and information:
    a. How many people are attending, and limits on how many may register.
    b. The ability to register for an event and selecting which of their children are attending.
    c. The ability to withdraw attendance from an event if already marked attending.
    d. Any other event or project information if present.
    e. Only admins should be able to see everyone in attendance.
2. Attendance can be marked through a process like UCF Here where they scan a code on the app under the event and it marks them as present at the event.
3. Attendance will need to be collected on a page reachable only by admins to be displayed on an iPad or similar device at the event. Parents who are not yet

registered will be told to register on their phones prior to reaching the device, and the page on the device should refresh when a new user is present with fallback search capabilities. Attendance is collected by the parent affirming they are attending using a button or checkbox.

4. Events are broken down into two categories: projects and events. Projects are a large, overarching objective that will take many people many hours to complete. These projects will be broken down into smaller events that will help contribute towards the end goal of the overall project.

5.1.5 Volunteer Solutions

1. Users have the option of also volunteering for events and optionally selecting project-based tasks they can help with for the event. These tasks will tally up points towards their work on the project and should be manually editable on the backend in the event they did not attend.
2. Project pages and the table of users on the admin panel should show a breakdown of volunteer hours for each project and user, with the ability to manually enter volunteer hours.
3. There should be a page for volunteers to check-in as well with whatever attendance solution we choose to use for the students. Their stats and hours will need to be tracked on the website as well. They should also be included in the event data and records after the event ends.
4. We could also have a way for volunteers to be added on the day of events at the location as well as allowing them to register for an account with the OMC at the end of the event. This could be useful if they wanted to volunteer at more events or become involved with other aspects of the OMC.

5.2 Hosting Platform

Deciding where and how to host a website greatly influences the quality, availability, accessibility, and type of resources and services available to the developers and clients. With an overall budget of approximately $15 a month, the hosting solutions were explored carefully. Website development will take place on the AWS platform but the clients have a few options to consider that will support the website.

5.2.1 Database Hosting

Databases can either be installed by the developers and maintenance carried out as automated scripts and carefully laid out instructions for the clients, or a fully-managed

database can be purchased as a service. Managed databases provide enhanced usability, scalability, and reliability for a premium. Examples of managed database features include load and storage space scaling, daily backups, and automatic failover to restore the database in the event of a hardware failure. Managed databases are costly and are not ideal for the scale of our website, however, automated backups are important in ensuring the website can be restored in the event of a failure or if the system is tampered with. Since our PostgreSQL instance will be running alongside the other website components, backups can be made at a system level that can encompass more than just the database.

5.2.2 Node.js Hosting

Node.js is traditionally hosted on Linux servers on virtual or dedicated hardware, or through serverless functions.

Serverless functions are a cloud service allowing an API or individual endpoints to be stored on the cloud where the code is suspended in memory only to use CPU resources when requested. The cost of serverless functions is dependent on how much memory your functions will idle on, how long they run for, and how much bandwidth they consume. Setting aside the cost analysis of sourcing a small server for the database and bandwidth estimates, there is no concrete benefit to using them for the website. Serverless functions require special considerations during the development process, especially for a system that must run continuously to send event notifications. The cost benefits often also mean higher latency, a specific concern of ours as mobile devices compound latency slowdowns. Aside from the general inappropriateness of serverless functions for the website, serverless is not directly compatible with websockets, a feature of interest for the attendance page.

Virtual hardware, in contrast to dedicated hardware, has the implication that a single server is hosting many different virtual machines that are rented out to different customers. You rent a specific amount of CPU cores, memory, storage, and are allowed to freely use those resources without any side-effects from the other virtual neighbors if the hosting provider is reputable. The network connection to the VM host machine is shared, but proper providers place speed limits on ports to prevent the VM from being starved of bandwidth due to the other customers. The use of virtual machines is more cost-effective, and are easier to upgrade or downgrade if the website needs to scale.

5.2.3 Storage Space

Node.js, PostgreSQL, and the uploaded user files all require volatile file storage. Storage is the cheapest of hardware resources but certain providers charge for storage in different ways. The typical use-case for uploading files is when parents want to submit a form for free or reduced lunch for approval by the administrators. Even if every user uploaded this form, and the document was large (e.g. 10MB or larger), it would take many multiples of the expected number of users to be a cause for concern if the documents were never deleted after approval. When storage is included with the cost of the virtual machine, it is usually a sufficiently large SSD. However, cloud providers that do not bundle in this cost often have many different kinds of storage options to consider.

5.2.4 Providers Compared

The two providers that we recommend are DigitalOcean and AWS based on price–performance. Other services such as Microsoft's Azure, Google's Cloud Compute, and standalone VPS providers such as Vultr, can be considered by the client as the website will be compatible with a typical Linux environment. Orlando Math Circle is a 501(c)(3) tax-exempt charitable organization and as such it may be worth reaching out to the sales teams of different hosting providers to determine if there are discounts or benefits that make clearer distinctions beyond the following analysis.

5.2.4.1 DigitalOcean

DigitalOcean is a developer favorite for its ease-of-use, reliability, and pricing that remains competitive against Amazon's AWS. They call their virtual machines droplets, and each has a standard hourly cost up to a flat monthly cap. There are different kinds of droplets determined by the number of CPU cores, GBs of memory, TBs of data transfer, and GBs of SSD storage for a given price point.

Droplets can be resized, or changed to a more or less powerful VM type, through the dashboard without training. It's not possible to downgrade to a VM with less storage, and upgrading to a VM with more storage is permanent, but resizing to a VM with the same amount of storage is reversible. This feature could allow a droplet to be resized in anticipation for a large event with ease for short periods of time.

Within the budget are five types of droplets provided by DigitalOcean depicted in Figure 1 from the DigitalOcean pricing page [1]. Benchmarks will be required to determine the

proper configuration that allows for the most concurrent users, however, the middle option of 2 vCPUs and 2 GB of RAM is likely sufficient for most workloads. A smaller droplet will likely be bottlenecked by either the single vCPU or the memory, and upgrading from these would be permanent as they have smaller SSD sizes.

VCPUs	Memory	Transfer	SSD Disk	Price
1vCPU	1 GB	1 TB	25 GB	$5/mo $0.007/hr
1 vCPU	2 GB	2 TB	50 GB	$10/mo $0.015/hr
2 vCPUs	2 GB	3 TB	60 GB	$15/mo $0.022/hr
3 vCPUs	1 GB	3 TB	60 GB	$15/mo $0.022/hr
1 vCPUs	3 GB	3 TB	60 GB	$15/mo $0.022/hr

Figure 1: DigitalOcean Standard Droplet Prices [1]

5.2.4.2 Amazon Web Services

Amazon Web Services (AWS) is the dominant cloud provider alongside Microsoft Azure and Google Cloud Compute for their array of developer services. AWS provides a high level of granularity in how its services are billed and consumed such as providing price reductions when sending data from one AWS service to another in the same region. There are hundreds of AWS services, and some services can be used in an equally large number of different configurations, locations, and networks on top of a complex policy system. AWS is vast enough that it offers certifications to those fluent in their ecosystem. Despite their dominance in the market, their dashboard is daunting and costs can often be hidden by many different types of egress data charges and configuration pitfalls.

AWS EC2 provides virtual machines in different families denoted by the prefix in the instance name based on their CPU architecture, workload optimization, or other features unique to the family. For the price range of this project, the t2 and t3 families have a few instance types that may be appropriate, however, costs are updated frequently and can be determined on the AWS Simple Monthly Calculator [2]. The t2 and t3 families also

utilize a feature called burstable CPU credits, a cost-saving feature that guarantees a baseline performance level that can increase temporarily by acquiring credits below a vCPU utilization threshold. For instance, a t3.small (2 vCPUs and 2 GB memory) is the balanced choice for this project and has a baseline performance per vCPU of 20% [3]. Credits are earned for this instance at a rate of 24 per hour, and when the 20% threshold is breached, a credit is consumed at the rate of 1 per minute to provide full utilization. An unlimited feature can be utilized and is enabled by default, which allows the VM to burst beyond the accrued credits by charging for any credits not made back within the hour. When disabled, if credits are not available, the performance of the instance is capped at the baseline. For a small website that will not receive a high number of users throughout the day, this feature is ideal as guaranteed performance instances cost more.

AWS bills EC2 VM on-demand instances monthly per hour used, or by reserving instances through payment plans on 1 or 3-year commitments [4]. On-demand instances allow the instance type to be changed as needed. Standard reserved instances allow you to vertically grow an instance within the same family, such as upgrading to t3.large from t3.small. Convertible instances provide a smaller discount but allow for the family type to be changed. With reserved instances it is not possible to change to a different kind of family, e.g. t3 to t2, and downgrading instances is not possible with either plan. Figure 2 illustrates the different kinds of pricing plans for a year contract. AWS does provide a similar service to DigitalOcean called AWS Lightsail that bundles in all necessary resources, but the plans only match DigitalOcean and skip the $15 range altogether.

T3.Small On-Demand Pricing 1-Year				
Payment Scheme	**Upfront Price**	**Monthly**	**1 Year Cost**	**Savings**
On-Demand	$0	$15.23	$182.76	0%
T3.Small Standard 1-Year Term				
Payment Scheme	**Upfront Price**	**Monthly**	**1 Year Cost**	**Savings**
No Upfront	$0	$9.49	$113.88	38%
Partial Upfront	$54	$4.53	$108.36	41%
All Upfront	$107	$0	$107	41%
T3.Small Convertible 1-Year Term				

Payment Scheme	Upfront Price	Monthly	1 Year Cost	Savings
No Upfront	$0	$10.95	$131.40	28%
Partial Upfront	$63	$5.18	$125.16	32%
All Upfront	$123	$0	$123	33%

Figure 2: t3.small EC2 instance 1-year pricing comparison.

5.2.5 Platform Recommendations

Our current recommendation is DigitalOcean as it outperforms AWS for monthly, on-demand billing plans because AWS only includes CPU and RAM in their pricing. For instance, a t3.small (2 vCPU and 2GB memory) instance costs $15.23 monthly and only allows full CPU utilization while spending credits accrued during low-load periods. Third-party benchmarks of the two companies' VMs determine that AWS has slightly better CPU and latency benchmarks, but the performance differences between both are negligible at this price point [3].

DigitalOcean advantages over AWS:

1. The dashboard is simple and does not require technical expertise to operate, monitor, or resize a VM.
2. DigitalOcean has best-in-class documentation that would prove valuable for educational purposes.
3. Price for performance, DigitalOcean outshines AWS.
4. AWS is more expensive without a reserved instance, and mixing many different AWS services would be needed to save the most money. The more AWS features used, the more intricate the website becomes, making maintenance harder.

AWS advantages over DigitalOcean:

1. Expansion of the website for new features, such as push notifications, would be cheaper within the AWS ecosystem. AWS Simple Email Service, while incredibly cheap, is free when the data comes from an EC2 VM.
2. New AWS accounts can use many AWS services for free or cheaply for an entire year after registration. DigitalOcean provides a large credit from the GitHub education pack, but it is not as exhaustive.
3. A reserved instance could save a considerable amount of money if the commitment is carefully considered.

4. AWS has more educational opportunities for students if they wish to learn more about the ecosystem.

DigitalOcean will cost $15 per month (not including emailing) while AWS will be approximately $17 or more monthly without commitments. Figure 3 illustrates the extra pricing resulting from data transfer and storage costs. Data transfer is free for the first GB each month and is not likely to be saturated by the website, however, the storage is costly. Parity with DigitalOcean's included space would be $21.23 a month. Even if a 1-year contract was used to reserve an instance, the monthly cost would be $15.43 a month with the same size SSD drive. Be that as it may, a 20GB EBS SSD would be sufficient for PostgreSQL, Node.js, and the operating system, allowing the rest of the space to be used by AWS S3 Buckets which cost $0.023 per GB resulting in an approximate $13 a month when considering data transfer. Importantly, the guaranteed value of the CPU cores is never higher than 20% load with AWS, which poses a performance risk without potentially spending more for unlimited boosting.

The cost benefits of AWS are not apparent without a commitment that may be dangerous to the clients if the expected number of users is not properly gauged or other features need to be included in that time frame. Backups for both providers are $0.05 per GB of each snapshot and is not a significant determining factor.

VM Instance	vCPU	RAM	Transfer	Storage	Cost
General Purpose Droplet	2 vCPUs	2 GB	3 TB	60 GB	$15/mo
AWS EC2 t3.small	2 vCPUs*	2 GB	$0.09 / GB**	$0.10 / GB***	$17+/mo

Figure 3: DigitalOcean vs AWS EC2 cost breakdown.

* Sustained usage of more than 20% of a vCPU core will require credits accrued during low usage or will be capped at 20% utilization.
** Only egress data is charged. The first GB of egress is free each month, making transfer free for the size of the application.
** Traditional server SSD drives are part of AWS EBS (Elastic Block Storage). S3 bucket storage is considerably cheaper and can be used for the file uploads.

5.3 Emailing

As a security feature and a notification medium, emailing will be the primary means of communication with users. Emails are used for confirming account registrations, resetting forgotten passwords, notifying users of upcoming events, and sending emails to groups of users from the admin panel. Sending emails is simple, but securely setting up an SMTP server is not. After reviewing various emailing services we will be developing the website using Amazon's Simple Email Service (SES), however Sendinblue, GSuite, and Mailgun were also investigated. The emailing templates will be mostly ubiquitous, thus the emailing provider used is one of the easier services to replace in the website if necessary, e.g. if the SES quota is unsatisfactory.

To quickly mention push-notifications, the clients do not wish for a progressive website (PWA) as the installation of the website app may be confusing to the parent demographic. To that end, a service worker could be made with the sole purpose of sending push notifications, however this feature is under consideration as a homerun requirement.

5.3.1 AWS Simple Email Service (SES)

Amazon's SES is a no-frills emailing service that provides attractive pay-as-you-use pricing and excellent deliverability [36]. Every 1,000 outgoing emails is $0.10 and data costs $0.12 per GB. Incoming data is charged in whole chunks of 256 KB at 1,000 per $0.09, or 256 MB per $0.09. Nonetheless, SES does not provide many features beyond those meant for managing email bounces, or emails which failed to send, and complaints. Notably, it is not easy to forward emails received by SES to an email client. If two-way communication is desired by the client from emails sent by the website, for all intents and purposes SES is not a workable solution.

5.3.1.1 Security Policies

To ensure that the emails have a high rate of deliverability, the following policies and settings must be configured on a domain that is verified with SES.

- **Domain-based Message Authentication, Reporting and Conformance (DMARC)**: DMARC is a protocol defined in a TXT record of the domain sending emails used to tell the receiving server what to do when either SPF or DKIM are not successful.

The first policy, none, tells the receiving server to accept the email even though verification was not total. The server can choose to honor this or not. The second policy, quarantine, tells the server to put the message in the spam folder. The third policy, reject, tells the server to block the message. The first two policies will show up in DMARC reports sent to an email defined in the policy to monitor issues or phishing attempts with the domain. Rejecting the message makes it difficult to diagnose false-positives; rejected emails do not send bounces to the mailing server. The emails ISPs send for DMARC reports could be annoying to the clients, and services exist to filter these emails outside of emergent circumstances or consolidate them, if necessary.

- **Sender Policy Framework (SPF)**: SPF is an anti-spoofing measure done by listing the mail servers in the DNS records of the domain. Servers receiving emails will match the MAIL FROM header with this DNS record on the domain. This header contains an email subdomain used to receive bounces and complaints, e.g. bounce@example.com. By default, AWS specifies its own MAIL FROM domain to provide partial validation, however, complete validation requires that the FROM address used to send emails contains the same domain as that in the MAIL FROM header. This subdomain can be installed using DNS records.
- **DomainKeys Identified Mail (DKIM)**: DKIM is an encrypted representation of an email stored in the header with a private key known only by the server, and a DNS record is made containing the public key so receiving servers can use to decrypt the header to determine trust. This ensures the message wasn't tampered with and further describes that it came from the appropriate source. AWS provides a free feature called Easy DKIM to sign emails sent by their service automatically.

5.3.1.2 Bounces and Complaints

Maintaining a healthy relationship with SES means ISPs are kept happy by not generating a lot of bounces or complaints from users. AWS will receive notifications when an email is hard bounced, soft bounced, or a complaint is received (marked as spam). Hard bounces are sent from an ISP when the email doesn't exist, and soft bounces are when there are temporary issues with sending the email, such as connection problems or a full inbox. Soft bounces may want to be retried, but AWS only generates a notification for them if it cannot be sent after a retrying for some time.

AWS tracks hard-bounces at a global level and both hard bounces and complaints at an account level. The former will still count towards the bounce rate of the account, whereas the second does not and can be queried against and emails manually added or removed. Both still count towards the number of emails sent, so they are not a sufficient means of protection, and neither handle excessive soft-bounces which can damage relationships with some ISPs.

5.3.1.3 Bounce and Complaint Strategy

As part of the process to request a quota for sending emails, it must be affirmed that there is a process in place to handle bounces and complaints. Similarly, the website must be provided for their review so they can see how it collects emails and if they are opt-in (we will be using double opt-in, as recommended). By default, bounces and complaints send emails to the account holder for someone to manually make the necessary user modifications, say through the admin panel. Ideally, the clients should not need to worry about this, so SES provides a means of directing these messages to their Simple Notification Service (SNS).

The delivery mechanisms for SNS vary, however, the most reliable method would be to send messages to the AWS Simple Queue Service (SQS). Every request to SQS is billed at 1,000,000 requests for $0.40, effectively free for this purpose. To illustrate this process, a diagram in Figure 4 shows the bounce and complaint flow. Once the server receives the bounce or complaint, a property on the user's account will be flipped depending on the type of response.

- In the event of a complaint, all notifications are disabled. The only emails the user would receive are transactional emails they themselves have generated.
- Soft bounces are less of a problem, but they may impact the site's reputation if they are a frequent occurrence. A configurable environment variable will be provided to set a threshold for when notification emails should be disabled for users with excessive email bouncing.
- Hard bounces can seriously damage OMC's reputation, thus hard bounces immediately invalidate the email and disallows future transactional emails to this email. The user will be required to verify a different email. This is likely to occur if someone accidentally enters an invalid email.

5.3.1.4 Approval Process

To leave the sandboxed SES environment, the website must be approved to use their service. A support case is opened titled SES Spending Limits and a lengthy form is filled out detailing the exact usage specifications of the website. AWS guards their sending reputation with extreme prejudice, and it is noted online that failure to include a field or inadequate descriptions on some fields can lead to multiple denied requests. In one way, this is a contract between the developers, AWS, and the OMC staff that will write emails in the admin panel to ensure the service is used in the way described.

The form includes sections that require we describe how we will be adhering to the AWS acceptable use policy, why we are sending emails, how we are systemically handling bounces and complaints, what our website domain is (so they can manually review how we collect emails), if we confirm the emails of users, and what limits we would like to receive. The requested limits need to be humble, as building trust on the platform is expected, then requests for higher limits can be made if necessary. The base limit is 200 emails in a 24 hour window, and for most days this is likely acceptable. A point of failure is likely to occur if the initial release of the website causes an influx of new registrations that overlaps with a near event. With an estimated total users of 130-150, at two emails per person the limit would be unsatisfactory. Thus a daily raise of 50-100 emails is not unreasonable, but potentially unnecessary if the first request is denied for further information.

The default per-second sending rate of 1 considers all recipients, so sending emails in bulk does not circumvent this limit. Using a queue, emails can be sent at a rate of 1 per second, however with latency and transactional emails this delay might become pronounced as more users join the app. Once a healthy record of production emails are sent, OMC may request this limit increased to, hypothetically, 5 per second and we will provide environment variables to take these limits into account. Asking for multiple increases simultaneously is likely to increase the odds of rejection.

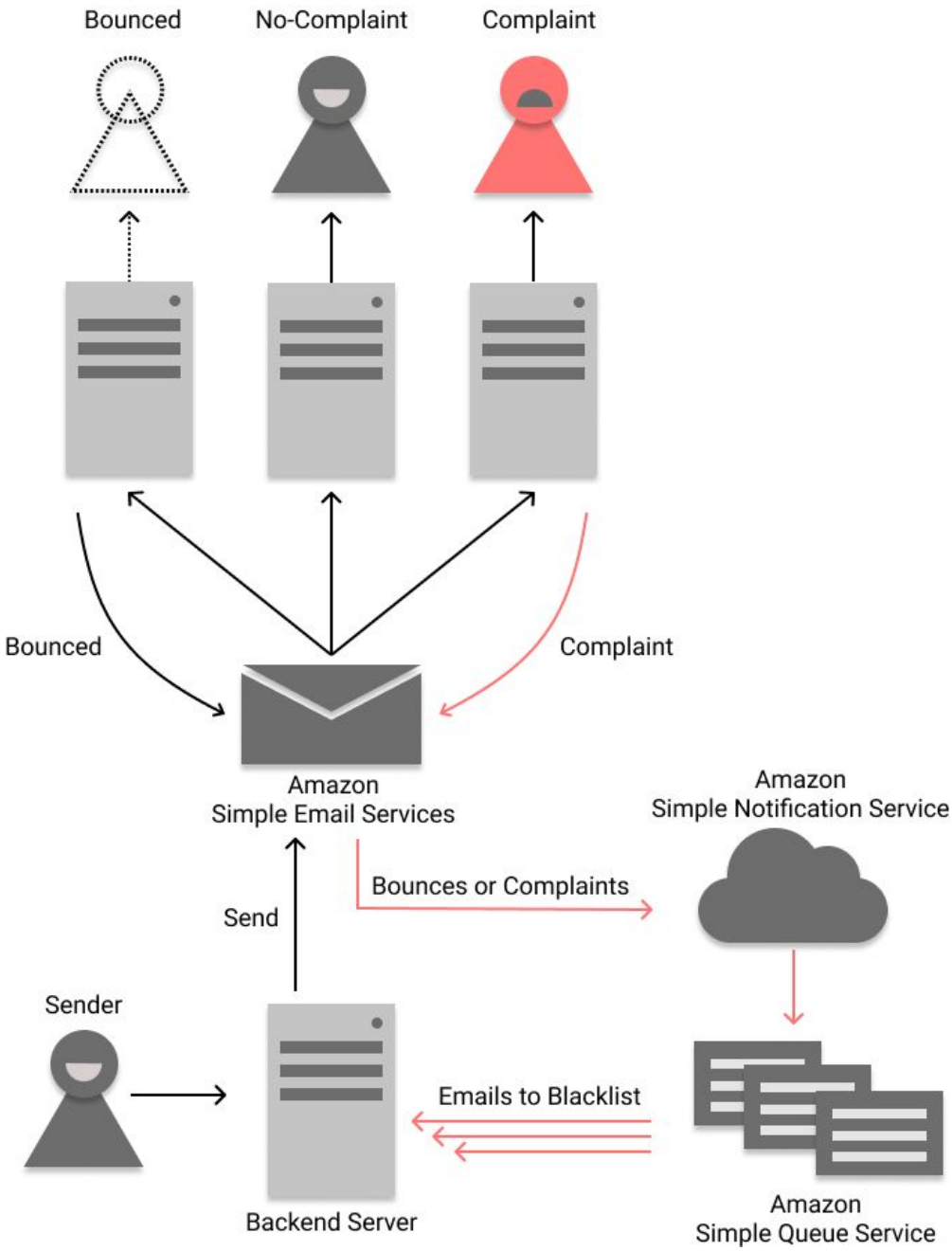


Figure 4: AWS bounce and complaint strategy transport.

5.3.1.5 Sending Emails

Using the AWS Node.js SDK, the two predominant methods that will be used are `SendEmail` and `SendEmailRaw`. The former is for simple emails, whereas the latter supports HTML as well as sending attachments. Both methods can send emails to individual recipients or in bulk to up to 50 people with both counting against the quota

depending on the number of recipients. In terms of notification emails, the website will need to be weary of not exceeding the daily or secondly quota limits for sending messages as well as not trying to send any single notification email to more than 50 people using one method. If a notification should go to more than 50 people, or the quoted per-second limit is below the number of recipients, the recipient list must be split and sent separately. AWS does not recommend sending emails in bulk, however. Further sections will discuss using queues to safely send emails from different processes without accidentally reaching the limits.

5.3.2 Alternative Providers

In the event that AWS SES does not prove ideal for either the client's needs or the approval process was unsatisfactory, the following alternatives provide nearly identical services, with the more expensive tiers offering advanced features such as mailing list management, easier processes for dealing with bounces and complaints, and analytics that would determine how engaged users are with the service.

- Sendinblue: Their service has a free tier of 300 emails per day, however it was not considered above AWS due to a large watermark they place in the emails.
- Mailgun: Popular emailing provider for developers with a flexible tier for $0.80 per 1,000 emails. They also include graphical analytics for seeing the percentages of emails missing users and offering insights on how to fix those problems.
- GSuite: Google Suite is $6.00 a month normally but free for nonprofits. They offer unlimited users with 30 GB of drive space each and multiple email addresses compatible with the website and the Gmail interface if two-way emailing is desired. Using GSuite would be ideal as AWS requires an approval process anyway, and while cheap is not free like GSuite for Nonprofits. While AWS can be installed entirely by us, interacting with GSuite is more of an endeavor on their part they may not want to utilize. In a similar vein, GSuite's primary purpose is not for transactional emails, so handling bounces and complaints may require more involvement.

5.4 Security

Imagine an attacker who knows that you utilize a certain online banking platform. They could embed a script in a website of theirs that communicates with your banking platform to initiate a bank transfer to their accounts. Perhaps you visit this website by clicking a link in your email, resulting in the malicious website initiating the bank transfer. From

the bank's point of view, you are a regular user since your browser retains the cookie for their site and thus it allows the request to go through. This is known as a cross-site request forgery (CSRF) attack [2]. Various security practices have been explored for our service to combat CSRF and other common vulnerabilities.

5.4.1 Password Hashing & Salting

One of the most fundamental security procedures is not storing user passwords in a database. Instead, hashes are used to make intentionally convoluted transformations to passwords with a secret only the server knows, and storing this transformation. Hashing algorithms transform data of arbitrary size to a fixed size, are only one-way, and should be infeasible to revert. The best hashing algorithms are the ones where brute-force is the only reasonable way of finding inputs. However, their fast and deterministic nature allows for methodical exploitation such as rainbow table attacks — a linked list of precomputed hashes for all characters up to a certain length. This can be made harder by salting the password first, or concatenating the password with a string of random characters. The salted password is then hashed and stored alongside the salt string. With hashing and salting, not only is the password never stored, its hash is random even if multiple users have the same password, greatly increasing the amount of effort needed to attack it.

Password hashing and salting can be achieved securely using the adaptive bcrypt function developed by Niels Provos and David Mazières in 1999 [40]. The function is based on the Blowfish cipher and added configurable rounds for increasing complexity as hardware improves. It is recommended to benchmark the hardware running bcrypt to determine an acceptable number of rounds whilst not unintentionally sacrificing users to long loading times. Figure 5 shows some sample code for testing bcrypt speed as a function of the number of rounds [2]. Two devices were tested and graphed in Figure 6.

```
const bcrypt = require('bcrypt');

const plainTextPassword1 = "DFGh5546*%^__90";

for (let saltRounds = 10; saltRounds < 21; saltRounds++) {
  console.time(`bcrypt | cost: ${saltRounds}, time to hash`);
  bcrypt.hashSync(plainTextPassword1, saltRounds);
  console.timeEnd(`bcrypt | cost: ${saltRounds}, time to hash`);
}
```

Figure 5: The Node.js testing code for Bcrypt rounds.

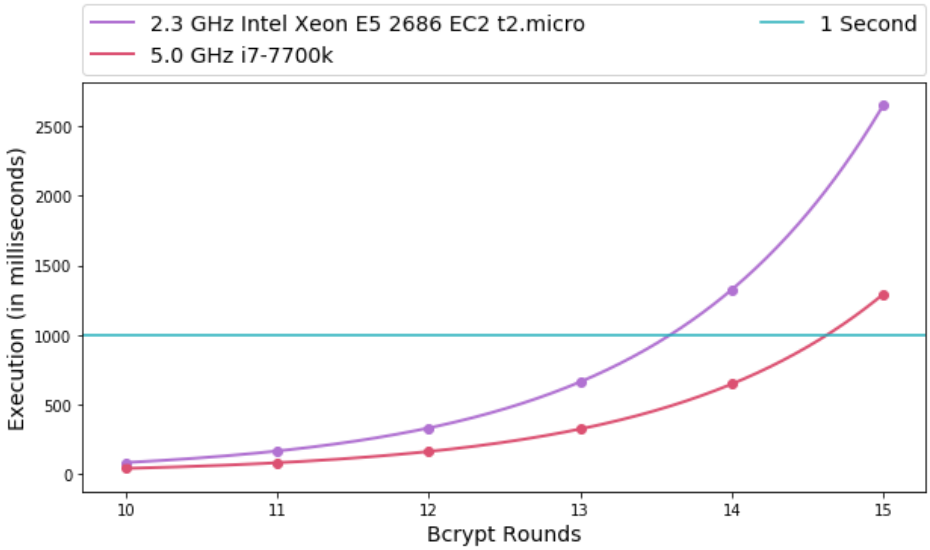


Figure 6: Exponential line fit showing the speed of two processors against bcrypt rounds.

It was conveyed by the clients that internet connectivity and latency are a concern for some event venues. If the registration and login process were only a fixed 1 second, users would be fine waiting momentarily with a loading indicator. However, the backend will also need to communicate with the database and send the response back to the user, all actions that accrue non-insignificant latency. We will be utilizing 12 rounds as the t2.micro will likely have a similar performance per-core as the production environment.

5.4.2 SSL Encryption

SSL (HTTP over SSL, or HTTPS) will encode all ingoing and outgoing traffic for the database, backend, and frontend server components on top of any communication with

external APIs such as PayPal. This prevents man-in-the-middle attacks which involve attackers positioning themselves between the requests sent between a user and the server, allowing them to intercept potentially sensitive information.

LetsEncrypt is a free service that provides a valid SSL certificate with its own certificate authority. The NGINX layer will be configured to provide SSL for the domain and the API subdomain. A cron task will need to be registered on the server that refreshes the certificate at the recommended 60-day interval before the 90-day certificate duration elapses.

5.4.3 Transmission, Cookies, and CSRF

During the login flow, if the server responds with an instruction to set a cookie, the browser will automatically send this cookie on subsequent requests to the domain where the cookie originated. This automatic browser authentication makes the server resources vulnerable to CSRF attacks. Prevention of CSRF attacks depends on the design of the system utilizing either a modification to authentication to not make it automatic or by not implicitly trusting cookies without a secondary token.

Cookies have attributes that limit their scope and locality: the secure flag restricts a cookie to only operate over SSL, the domain flag limits the cookie's use to a specific domain and its subdomains, httpOnly disallows JavaScript to interact with the cookie, and expires sets the time in which the cookie should be deleted. The enforcement of these and other attributes is up to the browser that the user is on with older browsers supporting fewer features, such as the newer httpOnly attribute. As such, relying on the browser of the user to use all modern security features is not sufficient. As mentioned, there are two primary defence mechanisms:

1. Store the token client-side in localStorage or as a cookie without httpOnly enabled. When authentication is necessary, retrieve the token with JavaScript and pass this information as a bearer header to the server. The server will also be set to not accept automatic cookies. Token transference should not occur in the URL to prevent it being intercepted, cached, or logged.

   This approach is common in single-page applications (SPAs), especially with JWTs, as the frontend is more likely to be decoupled from the backend or hosted statically. While cross-site scripting (XSS) attacks make it possible for injected

JavaScript to read localStorage or a cookie without the httpOnly attribute, CSRF can be defeated using XSS.

2. Have the backend instantiate the frontend rendering and inject the CSRF token, typically through a request middleware. This approach is not ideal for our design ideology as it makes the API not RESTful which may sacrifice the ease of maintainability, future developments, and educational value as coupling introduces complications.

We are going to be designing the website with the first approach in mind. Choosing reputable libraries that are known to be production-ready and not implicitly trusting any input from users is important to prevent XSS attacks that this approach accentuates.

5.4.4 CORS Policies

Cross-origin resource sharing is a set of headers that the server sends to the browser to instruct it to limit the availability of server resources and methods (GET, POST, DELETE, ...etc) to only the desired origin or domain. The backend will employ an Access-Control-Allow-Origin header set to the domain of the website to prevent browsers from accessing server resources on other domains. If the backend was to be expanded to connect to other services or open up its API, potentially for educational purposes, the backend would need to be modified to make this more permissive.

5.5 Authentication

Sessions are the dominant architecture for maintaining user state by storing a token or id after their successful authentication as a cookie on the user's device. This token is sent with requests to the backend server and is matched with a database row or record in memory to match the state. In practice, the token is often a hashed value with a secret only known to the server. However, the emergence of microservices, RESTful APIs, and serverless technologies challenge the popularity of sessions with JSON Web Tokens (JWTs).

JWTs retain the user state on the client-side instead of just an identifier. The common implementation contains a cryptographically hashed segment with the rest of the token claims visible. Tampering is prevented as the hashed component also includes the unhashed sections of the token. This implies that JWTs are stateless, or that there is no stored state on the backend necessary for authentication; if the token decodes properly then the claims within it are trustworthy. Stateless authentication is ideal for providing

single-use access to external microservices, but is harder to trust the claims when used similarly to sessions.

Whenever possible, sessions are the preferred technology. However, the popularity of JWTs have arisen due to complications, mechanical and philosophical, of trying to incorporate sessions into non-monolithic architectures. The common transport mechanism for sessions, httpOnly Cookies, are sent automatically on requests thus warranting a secondary means of preventing CSRF attacks. Anti-CSRF tokens are injected through server-side rendering into the frontend code of each page, something not possible with a decoupled REST API. JWTs also more closely adheres to the idea of statelessness, or that the API receives all necessary logic from the request itself, not from some saved state on the server. Ultimately, the use of JWTs in a stateful manner are controversial at best, but it most importantly allows for the greatest degree of freedom when expanding the website in the future.

5.5.1 JWT Revocation

Revoking individual JWTs is done by either storing the token in a blacklist table until its expiration date or by generating different secrets for every user. The second approach is ideal as retrieving the user is a necessary step during token creation, validation, and eventually passing user data to the requested route. One such implementation of using user information as part of the token's hashing secret is to concatenate the user's hashed and salted password with a universal secret in the environment variables. This would mean that changing the password of a user would immediately revoke all tokens for that user.

Implementing a logout feature for an entire account could also be done by adding a date field to the user's account that is populated with the current time whenever a user requests to logout of all devices. Tokens with an issued-at field before this date will be rejected.

5.5.2 JWT Structure

While encoded, the JWT is stored as a cookie on the client's device and is transferred as a header during server requests requiring authentication. An example JWT can be seen in Figure 7.

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.

```
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ik
  pvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.
 QUwp8Yc679Jv0sIMLALlmwnVRzHps9sihKw9qt
```

Figure 7: Example JSON Web Token.

A JWT contains a header, payload, and signature components that are base64 URL encoded and period-delimited. The header component describes the algorithm used to encode the type of JWT used, and the payload contains user-specific claims or other necessary information such as their id, when it was issued, and its expiration.

1. Header (encryption algorithm):
   `{ "alg":"HS256", “typ”: “JWT” }`
2. Payload (transfer data):
   `{ “id”: 1, “iat”: 1583014539, “exp”: 1585606539 }`
3. Signature (hashed header, payload, secret + user random value):
   `HMACSHA256(base64(header) + “.” + base64(payload), “256-bit-secret”)`

5.6 Authorization

Endpoints will require determining if the authenticated user has the correct permissions to perform the requested action. Actions will also need to limit their scope to users who should not overstep the permissions granted to them. For instance, primary users should be able to read, update, or delete their own registration to an event, but only admins should be able to edit registrations they don’t own. Each event will also contain metadata describing attributes necessary for user registrations, such as specifying the age ranges of children who may attend the event.

Role-based access control (RBAC) is the popular approach that provides named roles that describe access to resources. Endpoints can utilize RBAC through the Access Control library [6]. Since most of the permissions related to events will be metadata driven, an attribute-based access control (ABAC) library was sought after. ABAC is a more encompassing approach to access control by allowing attributes or circumstances of the actors, resources, and environment variables to be used to make policies that govern how resources can be interacted with. An appropriate RBAC policy for attorneys may, for example, only allow access to files when they are listed as a client of the actor before 5pm. While powerful, ABAC is much more difficult to implement as there is a layer of translation necessary from the abstract policy structure to the resource structures, which

are often translations themselves thanks to database ORMs. RBAC can be used much more simply using guards, a type of gatekeeping middleware discussed in the backend research section, and a list of roles on user entities, and attribute-based permissions can be checked manually as needed.

The Access Control library is used by granting, or denying, if an actor can create, read, update, or delete resources as well as if the actor needs to own the resource, or if the resource can be owned by anyone. Figure 8 shows how to describe roles and how to use the library to check if the permissions are granted.

```ts
export enum Role {
  Admin = 'Admin',
  Educator = 'Educator',
  Volunteer = 'Volunteer',
  Verified = 'Verified',
  Unverified = 'Unverified',
}

export const accessControl = new AccessControl();

accessControl
  .grant(Role.Unverified)
  .readAny(['event', 'project'], ['*'])
  .grant(Role.Verified)
  .extend(Role.Unverified)
  .createOwn(['event-registration', 'payment'], ['*'])
  .readOwn(['event-registration', 'payment', 'event-attendance'], ['*'])
  .updateOwn('event-registration', ['*'])
  .deleteOwn('event-registration', ['*'])
  .grant([Role.Volunteer, Role.Educator])
  .extend(Role.Verified)
  .grant(Role.Admin)
  .extend(Role.Educator)
  .createAny(['event', 'payment', 'project', 'event-attendance'], ['*'])
  .updateAny(['event', 'payment', 'project', 'event-attendance'], ['*'])
  .deleteAny(['event', 'payment', 'project', 'event-attendance'], ['*']);

const { granted: example1 } = accessControl.can('Admin').updateAny('event');
// example1 === True

const { granted: example2 } = accessControl.can('Volunteer').updateAny('event');
// example2 === false
```

Figure 8: AccessControl example roles, role assignment, and permission checking.

5.7 Database

The choice of database has important ramifications in underlying data structures of the application. The two types of databases are non-relational databases, colloquially called NoSQL databases, and relational, or SQL, databases. The distinction between the two is that NoSQL databases can use other querying languages besides SQL.

5.7.1 Non-Relational Database

Non-relational databases were originally developed to overcome the shortcomings of relational databases, specifically their scalability, data structures, and performance. These databases avoid the main idea of relational databases, their rows and columns table approach. Instead they focus on giving the database flexibility by avoiding implementation of a specific rigid structure that a database has to adhere to. This gives a database administrator the ability to create a database that is optimized for the different kinds of data that needs to be stored. Overall, the general theme of these databases is that they do not use any kind of relational model to connect the data.

These databases excel with scalability and performance. Non-relational databases have an impressive amount of space for their data. Due to their design, the data expands horizontally instead of vertically, this allows them to spread data across servers much more easily than with relational databases, improving space and costs [7]. The maximum scale with the most popular options for data structures, JSON document, column-family, and key/value, actually goes to the range of petabytes. Most non-relational database systems have the data stored as JSON documents which is where the improvement in performance comes in when the database is queried. Having the data already in JSON formatting allows for a more efficient process when sending information from the database to a website through JSON packets.

The issues that come up with non-relational databases are the lack of consistency between how to use the database (indexing) and the difficulty if the data needs to have relationships. While the variety of data structures offers flexibility, it also can make the system a bit more confusing and harder to accomplish the database's goals. One of the bigger factors in this is the difficulty with indexing in the non-relational database. While indexing is supported, since there are not concrete table structures in these databases the implementation can require more effort. The difficulty also comes from the fact that indexing can be implemented in various ways throughout the database, depending on the

variety of data structures that are used and how to achieve the proper indexing that is needed with each different structure. These details also come into play with the ability to use relational objects in a non-relational database. Again, it is something that is possible to do, however most of the databases offer little to no built in support for the process. The issue comes from the difficulty to implement the relations without the uniform table structures that relational databases were built with.

5.7.2 Relational Database

Relational databases are databases that store data using a relational model. They use tables as their data structure that use rows and columns to store and organize all of the data properly. Each row is its own record and each column holds a different field relating to the record. Therefore, each addition to the database adds a new row and the table scales vertically down. The relational model is what makes these databases the preferred choice for so many years. It allows for different tables in the database to have shared data in a secure and efficient way. This feature seems small at first glance, but being able to build a database with related tables and data allows for accurate and efficient data manipulation.

The relational database is popular due to its consistency and efficiency. The structure of the data lends itself toward a very cohesive collection of information. There are not any worries about lag when updating data, which is a very important upgrade when comparing between non-relational databases that may need some time when large amounts of data is added or accessed [8]. Also, with the relations it is simple to keep data points connected which allows for the database to pull or add connected components all at once. This prevents data being left in when its parent record has been deleted for instance, keeping all of the tables updated with current and accurate data. Having reliability with the database also leads to much more confident and efficient data manipulation. The relational database has useful functions and procedures that make updating or accessing data very simple. The uniform indexing allows for easy access to specific records. Procedures are easy to use and can be stored for those that are going to be repeated over and over again. Constraints and checks can be placed on certain data to ensure there are not any conflicts. The relational aspect adds so many layers that give the database a very user and data friendly interface.

The issues with a relational database are where the non-relational database excels. The data structures that give it its efficiency take away from its scalability and performance. Since the tables scale vertically, if the database is housed on a single server it requires

more and more power as the demand from the database grows. While it is still possible to service the database across multiple servers as it scales, it is much more costly to do so and it risks certain features between tables being lost with the data being spread between different servers [7]. The table structures also result in a decrease in performance when it comes to queries from the API. While queries within the database are strong and simple, transferring the necessary information in the necessary format requires some data manipulation. Specifically, when a record that is a part of a chain of relations is needed, the database engine needs to "join" these tables together to get all of the necessary data from the records that need to be put together and sent off. In addition to this, the data is not stored in the same format that it is sent to the website in, so the rows and columns need to be sent as JSON blobs which also takes a hit at the overall efficiency of the system.

5.7.3 MySQL Overview

MySQL is a Relational Database Management System (RDBMS) that is widely used and one of the most popular choices among relational database systems. It is considered to be the most famous large-scale database system. This gives MySQL a very large community with access to plenty of quality information and assistance that can be helpful when developing a database with it. Another important outside factor to point out is that it is the most commonly chosen database system for web development projects. This is because MySQL is known to be the most efficient when it comes to a large quantity of reads coming in, which is common to website usage. In addition to its efficiency, the database is also known to be highly secure as is and also offers lots of additional security features which is a big factor when it comes to developing a website that also needs to be as secure as possible.

There are downsides to MySQL though, especially when in comparison to a very advanced system like PostgreSQL. While MySQL is very efficient with read queries, the trade off comes with its inefficiency with write requests which it handles much slower, thus impacting its ability to handle concurrency. This could be an issue with our website as we foresee a large amount of writing to the database happening with its everyday usage. Another issue is that its only NoSQL support is JSON types and while this is not a major pressing issue for a relational database, the more options and support offered is always helpful even if it is not always needed. Lastly, MySQL does not use synchronous replication, with a second database synced to the first one, which gives way to a small possibility of data loss and while it is very small, any amount of loss is something that needs to be taken into consideration.

5.7.4 PostgreSQL Overview

PostgreSQL is an Object-Relational Database Management System (ORDBMS) that is a more recent addition among choices for relational database systems. It is “considered the most advanced open-source relational database in the world” [9]. It was developed to be extensibile, scalable, and more SQL compliant, while preserving its data integrity. PostgreSQL also has the addition of being object-relational which gives it a variety of very useful additional features such as table inheritance, intersection, and exceptions as well as advanced data types, specifically its fully defined boolean data type. Another important data type to note is its JSONB data type which allows for data to be stored in JSON blobs within the relational database. This is so important because it allows for the system to run the reliable relational database structure while improving the performance of outside queries with its JSON blobs. These JSONB data types also support indexing which improves the system even further and allows for faster access to the data.

Additional features that we took note of were its usage of synchronous replication and some key options that could clearly be useful to our project. These options were the CASCADE query which gives you the ability to drop every instance of data that is a dependency of the original data when you are deleting it. This could be a major positive when we have our database developed as we are planning on having a lot of dependent pieces, and being able to keep track of them efficiently will help us keep updated and cohesive tables. The other additional option was for individual tables to have a CHECK constraint which can be used, for example, to check a person’s age using other data in the same table before allowing a certain action to happen. We naturally can see ourselves using this constraint when we are managing user accounts in order to make sure our product continues to focus on being COPPA compliant when working on developing it for children. Also, with so much information and reliance on the database, having the synchronous replication, which has a “master” database synced with a “slave” database, provides an extra level of security knowing that both databases have to crash for data to be lost. Among these other functionalities of PostgreSQL, we did take note of the fact that Netflix uses it, which is what we have designed our base account setup off of.

There are a few downsides to be noted with using PostgreSQL. The deficiencies compared directly to MySQL are the lack of popularity with the database, which leads to a smaller support community behind the system and more reliance on ourselves to solve possible problems. Also, there is a steeper learning curve with PostgreSQL, partially due to this smaller community and partially to the additional functionalities of it. These

additional functionalities also bring about the issue of getting extra features in exchange for a decrease in speed, which is something that we will need to consider if we cho

5.7.5 Final Decision

For the database, we decided to use PostgreSQL. After going through the requirements for the website we saw that the way we needed to set up the database would require a lot of relations between the user information. This made it easy to eliminate a non-relational database such as MongoDB as it did not have the features that we needed to develop the database. While it was simple to choose a relational database, it took a bit more research to choose between the various ones that are available.

The preference for relational databases between web developers at the moment is either MySQL or PostgreSQL. There are a lot of similarities between the two databases, in fact on a surface level they are seemingly equivalent. The differences between the two databases comes in when you get into the more specialized features that are available to enhance the usage for administrators. The cons section for PostgreSQL is smaller and that is because it was honestly hard to find a large amount of downsides to using the database, especially when compared directly with MySQL. This was a large deciding factor for why we went with PostgreSQL. In addition to having fewer downsides compared to other relational databases, PostgreSQL actually makes up for some of the shortcomings of relational databases with its non-relational features. This gave us the opportunity to get the accuracy and relationships we wanted without sacrificing as much of the performance.

In addition to PostgreSQL's non-relational features, there are many more features that PostgreSQL has over MySQL that made the choice more clear [10]. Some of the features that we will likely use that are not supported in MySQL are features like more advanced data types, a clear Boolean type, more detailed output from the EXPLAIN query, table inheritance, intersection, and exceptions. We saw two options being immediately useful to our project development. The first was the CASCADE option that allows you to drop dependencies when an entity is deleted from a table which we can use to keep the integrity of the database intact due to the large amount of relations. The other was the CHECK option that acts as a constraint for certain variables which we can use when we need to check to make sure a user is over 13 before giving them access to certain functionalities of the website.

5.8 Backend Technologies

We constructed a list of potential backend languages and their respective frameworks once our requirements list was formalized. Some of our major areas of focus when making this decision were ease of development and versatility of the languages. We also valued other key aspects like performance for ensuring a respectable number of concurrent users for our price range and easy integration with our other technologies. The languages we had been looking more closely at for our research were Node.js, Python, PHP, and Java.

5.8.1 Node.js

Node.js is a cross-platform and open-source JavaScript runtime environment that executes Javascript code outside of a browser [11]. It uses the V8 JavaScript engine used by Chrome, the forerunner in development for implementing new JavaScript features and performance improvements. As a higher-level language, Node.js sets itself apart from other languages by being predominantly single-threaded but with the ability to efficiently run tasks in a loop. When Node.js is told to perform expensive operations, such as I/O operations by accessing the filesystem, instead of blocking the main thread and wasting CPU cycles, Node.js will ensure the main thread is still usable and wait for the right time to perform the task. As depicted in Figure 9, the event loop is a process of enqueuing and intelligent dequeuing from a task scheduler. The order in which tasks are performed depends on the state of the loop and the current phase, such as timers, waiting on I/O, idling, waiting for callbacks, and so forth.

Conversely, Node.js can spawn and manipulate threads, useful for performing expensive or crash-prone tasks in an entirely separate process, however the main thread is still run in one thread. This allows for thousands of concurrent connections without the user needing to manage thread concurrency, and Node.js is also able to be run in thread pools using tools like PM2 which help manage the server's runtime.

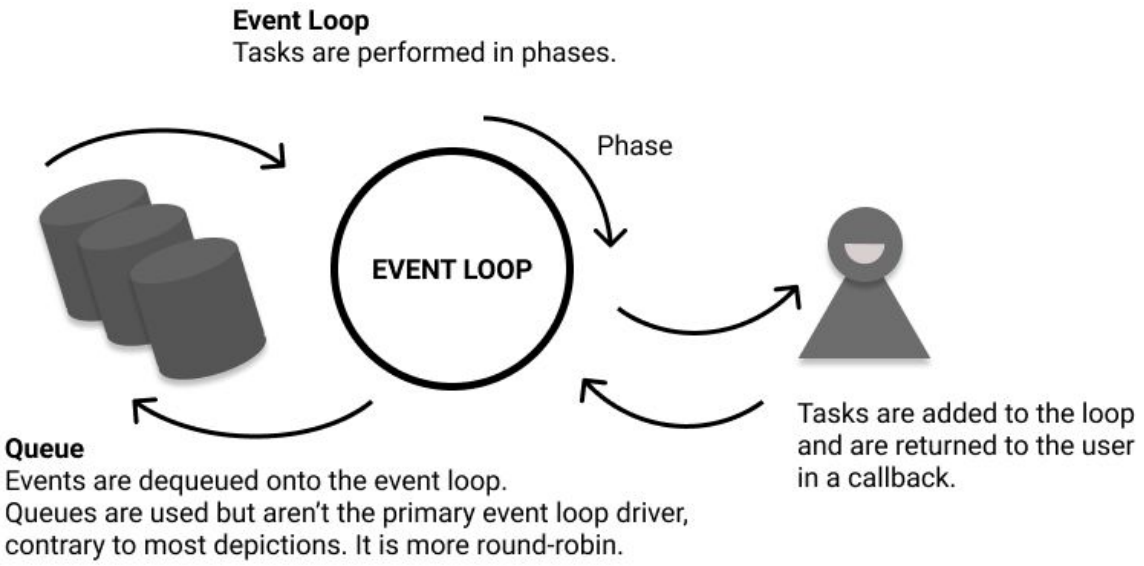


Figure 9: A typical rudimentary diagram for the Node.js event loop. In essence, users request a task be completed, and a callback is returned when it is finished.

Our clients want the code we make to be open-source, approachable, and of educational benefit to their students skilled at, and with interests in, programming without sacrificing on the feature set. Ultimately, dedicated students will be able to propose and implement new features to the website. This also favors Node.js as it means that both the frontend and backend can be developed exclusively in JavaScript. If it makes our lives as developers easier, it will also help the students who wish to learn from the project by not having to learn two languages to understand how the website operates.

Node.js is also a very popular framework for developing the backends of websites. Not only does this provide us with the largest package repository in the world with over 300,000 packages in NPM repository, but the number of tutorials and best-practices for each feature have been well-documented many times over. Node.js comes with the Node Package Manager (npm) allowing easy installations of packages and sharing package dependencies through version control in the `package.json` files. When installing a repository is necessary, it's as easy as `npm install`.

5.8.2 Asynchronous Programming

Node.js has the ability to utilize standard callbacks, promises, and syntactic sugar on top of promises using the async-await keywords, shown in Figure 10. Promises are an affirmation from the event loop that the task will be completed, much like an "I owe

you", which will either resolve or reject depending on if the task was successful or failed. The async-await keywords are merely syntactic sugar to resolve things that need to be run asynchronously automatically. The `promise.all()` function is useful for telling Node.js that both can occur simultaneously, but the main thread should resume when both finish successfully (or the first one fails).

```
async runAsync() {
  await Promise.resolve('Hello World!');

  await Promise.all([
    Promise.resolve('First async')
    Promise.resolve('Second async')
  ])
}
```

Figure 10: Example asynchronous function as syntactic sugar over promises.

5.9 Backend Framework — Nest.js

Nest.js is a meta-framework with a modular architecture incorporating Express, a popular API routing framework for Node.js [12, 13]. It was designed after the Angular frontend framework to provide a structurally opinionated framework with a high degree of versatility while abstracting away most of the Express layers. A few features that Nest.js borrows from Angular are its modular file structure, the injection system, and the liberal use of decorators. It has documentation on techniques such as authentication, database utilization, configuration management, input validation, task scheduling, security, queuing, file uploading, websockets, and others.

5.9.1 Decorators

Nearly all features of Nest.js are enhanced through the use of decorators — a stage 2 proposed JavaScript feature set to become part of the native language [14]. Since Node.js operates on the server it does not need code to be transpiled to support potentially outdated browsers. Decorators are higher-order functions that can be used to hook into class definitions, properties, methods, accessors, and parameters and create abstractions as seen in Figure 11. Much of Nest.js is designed to utilize other libraries and augment behaviors through its use of decorators.

```
function CapsLock() {
  return function(target: Object, key: string | symbol, descriptor: PropertyDescriptor) {
    const original = descriptor.value;

    descriptor.value = function() {
      const result: string = original.apply(this);

      return result.toUpperCase();
    };
    return descriptor;
  };
}

class Example {
  @CapsLock()
  helloWorld() {
    return `Hello World!`;
  }
}

const example = new Example();
console.log(example.helloWorld()); // Prints HELLO WORLD!
```

Figure 11: A simple decorator that transforms a return value into uppercase.

The ability to develop custom decorators is provided through Nest.js, but their use is limited as decorators can lead to excessive amounts of abstraction if not used correctly. Two use-cases where custom decorators prove useful is with authentication and validation. During authentication, the user is retrieved and appended to the request object to be used by the route. A custom user decorator can extract the `req.user` property as an argument in a controller, such as `@Usr() user: User`. Secondly, validation is done by describing a schema acceptable for a route through decorators. Sometimes this schema requires custom logic that is either dynamic, such as being dependent on information within the database, or requires custom context not provided by the default validation logic via custom decorators.

5.9.2 Modules

The architectural unit of Nest.js is the module, a self-enclosed structure that creates a grid of code units allowing for a highly organized project structure. The module system begins with a bootstrapping function seen in figure 12 that detects the presence of Express and allows for its configuration. The bootstrapping function incorporates a root module, typically called the AppModule, that serves as the root of a tree of all modules. The AppModule file by standard consists of only importing other modules, though if the app were small enough it could consist of only this module. CORS, global middleware, or

pipeline related logics can also be included in the bootstrap function before the listen method is called starting the application.

```
async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  const config: ConfigService = app.get(ConfigService);

  await app.listen(config.get<number>('PORT'));
}
bootstrap();
```

Figure 12: Nest constructing an Express instance on a configurable port.

Modules consist solely of a decorator and a class. The decorator has 4 constituent components: providers, controllers, imports, and exports; an example of which is seen in Figure 13.

1. **Imports** are a list of imported modules that export providers needed by this module.
2. **Exports** are a list of modules that are meant to be imported by other modules.
3. **Providers** are a list of services to be instantiated and used by controllers, other modules, or other providers. These are the files that make up most of the logic of the backend
4. **Controllers** are a list of controller files, or routing classes, that make up the API.

```
@Module({
  imports: [TypeOrmModule.forFeature([User])],
  providers: [UserService],
  controllers: [UserController],
  exports: [UserService]
})
export class UserModule {}
```

Figure 13: A Nest.js module for a typical entity.

5.9.3 Controllers

Controllers are classes decorated in such a way that they dynamically build API endpoints through the class-level controller decorator and individual method decorators that wrap Express. It is strongly recommended that only logic related to input validation, routing, API descriptions (such as with Swagger), and permission handling go in controller files.

Each controller function will require a method decorator corresponding to a standard HTTP request method. These methods return a default status code of `200 OK` by default, except for the `@Post()` method which returns `201 CREATED`. Changing the default status code returned can be done by the `@HttpCode()` decorator, which takes a single number input. Errors will always override the default status code.

Method	Use Case
`@Get()`	The get method retrieves, or gets, a resource or resources. This method should not modify data in any way, and sending body data is non-standard and should be avoided. There are usually multiples of this method that retrieve either singular entities, a list of entities, or entities by a given context as needed by the client. For example, if you had thousands of users, you would not wish to retrieve all of them simultaneously, you would have a get method which paginates them into blocks.
`@Post()`	The method used to create resources. It is standard that this method returns unique entries for each repeated request.
`@Put()`	Put is a controversial method as it is often used incorrectly. It is meant to update or create a resource if it is non-existent, commonly called an "upsert" method. Unlike post, if an entity exists already at this resource it is updated and not duplicated.
`@Patch()`	Patch is meant as an update method. It will not create an entity if it does not exist, and should return a `404 NOT FOUND` for missing entities.
`@Delete()`	Removes an entity from the database if it exists. It should return a success status if a deletion occurs, or a `404` status if the entity is missing.

There are other decorators used for browser-specific methods in the documentation. As seen in Figure 14, the endpoint for logging in would be `/auth/login`. The method

decorators are highly versatile and support describing parameters within the method they decorate. Parameters can be accessed using multiple different ways.

Decorator	Use Case
`@Param() params`	Captures all parameters as an object. Params need to be described using a colon within the method URL string, e.g. `GET /user/:id`
`@Param('id') id: number`	The param decorator can also accept an explicit parameter name and can be repeated multiple times.
`@Query() params`	Captures all query parameters as an object. A common example is paginating a large list of entities, e.g. `GET /user?take=10&skip=10`
`@Query('take') take: number`	Query also supports explicitly defining individual parameters.
`@Body() body`	Captures all of the body sent to the request as JSON
`@Body('name') name: string`	Captures specific parts attributes of the body.

Other commonly used decorators are the custom `@Usr()` decorator, which returns a reference to the user object passed from the authentication process. The `@Req()`, and `@Res()` decorators pass references to the request and response express arguments if they are needed, for example when writing custom body parsers or other decorators.

```ts
@Controller("auth")
export class AuthController {
  constructor(private readonly authService: AuthService) {}

  @UseGuards(LocalGuard)
  @Post("login")
  login(@Usr() user: User) {
    return this.authService.login(user);
  }

  @UseGuards(JwtGuard)
  @Get('profile')
  getProfile(@Usr() user: User) {
    return user;
  }
}
```

Figure 14: Controllers route through decorators and utilize services.

5.9.4 Providers

Providers can consist of services, database repositories, factories, and other logic or resource providing entities. The major differentiator between providers and regular classes is that they are denoted by the `@Injectable()` decorator, meaning they can be injected into controllers or other providers to mesh together. Providers are resolved within other files by including their type within the constructor seen in Figure 14 which Nest.js will scan and resolve the dependency injection. There are many niche features of dependency injection with the use of dependency injection tokens, or overriding providers with other classes or values [15].

Services are the most common type of provider used within Nest.js. A service is an abstract idea, but it should service a particular entity of function. A user service should provide all of the necessary CRUD (create, read, update, destroy) actions pertaining to users by interacting with the database directly, whereas an authentication service can utilize the user service to retrieve a user, but it should not directly communicate with the database. This separation of powers leads to cleaner code without repetitions. This is illustrated with an example of a rudimentary authentication service that takes an email and password and attempts to match it to a user in Figure 16.

```ts
@Injectable()
export class AuthService {
  constructor(private readonly userService: UserService) {}

  async login(email: string, password: string) {
    const user = await this.userService.findOne({ email });

    // This example would not work.
    return user && user.password_hash === password
     ? user
     : null;
  }
}
```

Figure 16: Services perform DRY operations, typically as CRUD actions on an entity or perform specialized but related tasks.

5.9.5 Exception Filters

Handling exceptions descriptively in the backend is fundamental for frontend developers trying to craft descriptive and informative errors for their users. This need not necessarily be by passing specific textual messages to the client's device, but is usually a combination of descriptive errors without potentially leaking or hinting at sensitive information, and HTTP status codes.

Nest.js provides a global exception filter that catches all unhandled errors within the application and returns a descriptive error to the user as seen in Figures 17 and 18. The global exception filter only handles exceptions thrown by errors inheriting the HttpException class. Custom exception filters can, and should, be made for things like database exceptions. Code can thus be written in such a way that an error terminates the call chain, and exception handling within methods will only be necessary for things such as optional behaviors, intended absences of entities, or fallback logic.

```ts
@Injectable()
export class UserService {
  async findOne(id: number) {
    throw new NotFoundException('Some example message');
  }
}
```

Figure 17: An example of code throwing a HTTP error.

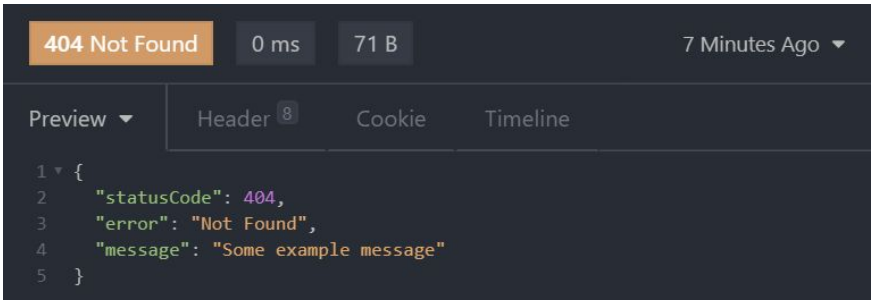


Figure 18: Errors using a class inheriting HttpException are caught automatically.

Construction of the backend will still require care to ensure errors thrown are instances of HttpException. Libraries Nest.js extends often do not throw these kinds of errors and will need exception filters written for them. Networking and critical errors in the state of the backend will return as a `500 Internal Server Error` by default to prevent a hanging request. Exception filters are added through the `@UseFilters` decorator, used on classes or methods, or in the Nest.js bootstrap function via `app.useGlobalFilters()` and passing a new instance of the filter class.

The list of HTTP status codes, what they are named, when they should be used, and how they should be handled by the client's device are listed in Mozilla Developer docs [16]. Common errors include successes 200-299, client errors 400-499, and server errors 500-599. Custom status codes can be utilized though not it is not recommended to deviate from the standard.

5.9.6 Middleware, Pipes, Guards, and Interceptors

Nest provides a few decorators that allow for logic to hook into behaviors. Each of them have situational uses and they typically have significant overlapping functionalities.

Middleware are injectable classes that have access to the Express request and response objects before the controller is called [17]. They have the capability to mutate the request and response objects, execute any code, pass control to the next middleware, or stop the request-response cycle if necessary. They are applied to modules either by specific routes or methods with wildcard support using a special configure method that can be given to modules.

Interceptors are highly capable injectable classes that allow for logic to be handled before a method similar to middleware, or afterwards to transform the output from the call [18]. This may mean transforming errors, including extra behaviors, or completely overriding the function of a method, such as with caching or dynamic validation. Unlike middleware, interceptors can be bound globally to the application using `app.useGlobalIncerceptors()` within the bootstrap function, or to individual controllers. Interceptors are highly versatile tools for performance analysis, logging, and throttling.

Pipes are injectable classes with a specialized role of transforming or validating data [19]. One common use for pipes is for transforming parameters from strings to numbers. The route `/user/1` has the 1 typically expressed as a string, since URLs are considered strings. This may annoy certain functions expecting numbers, and complicates TypeScript typings. Nest.js provides a ParseIntPipe for this purpose, along with a ValidationPipe covered in the validation section.

Guards are specialized injectable classes that enforce authentication strategies for the sole purpose of blocking or permitting requests [20]. Much of the authentication logic boils down to constructing guards. The JWT strategy that decodes a token and checks if a user exists, and if so, attaching them to the Express request object is called because of a guard. Guards make authentication and permissions simple to utilize within controllers without needing to write extra code shown in Figure 19. For instance, disallowing access to a `DELETE` route can be done by the built-in guard. A custom permissions guard may check if a user has sufficient permissions to access a resource.

```
@Controller('user')
export class UserController {
  constructor(private readonly accountService: AccountService) {}

  @UseGuards(JwtGuard)
  @Get('example')
  findOne() {
    return this.accountService.findOne(1);
  }
}
```

Figure 19: An example user route requiring basic authentication for a route.

5.9.7 Validation

Data sent to the backend is untrustworthy and should be validated. While MikroORM provides SQL injection prevention, the application may perform unnecessary work if the initial request is not written properly. To prevent needing to check parameters for their existence and validity within controllers, this can be done automatically through a validation pipe provided by Nest.js thanks to the Class-Validator library [21, 22]. The ValidationPipe can be initialized globally with a few different configurations in the bootstrap file illustrated in Figure 20.

ValidationPipe Options	Explanation
`disableErrorMessages`	When validation fails and a `400 Bad Request Error` is thrown, by default class-validator will also submit a list of errors detailing what was wrong with the request. This message is highly verbose and not likely necessary or useful in production.
`whitelist`	Properties without a decorator in a DTO class are simply ignored and are stripped from the validated object.
`forbidNonWhitelisted`	If whitelist is enabled, this option will throw a `400 Bad Request Error` if an unknown property is encountered instead of stripping them.
`transform`	Converts the plain javascript object into the instance of the DTO class with the proper typings.

```ts
async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  const config: ConfigService = app.get(ConfigService);

  app.useGlobalPipes(new ValidationPipe({
    disableErrorMessages: false,
    whitelist: true,
    forbidNonWhitelisted: false, // Throws an error on unexpected properties
    transform: true,
  }))

  await app.listen(config.get<number>('PORT'));
}
bootstrap();
```

Figure 20: A validation pipe with the major options set.

Once the global validation pipe is included in the bootstrap function it can be utilized in constructors by providing a type for the object returned by `@Param()`, `@Query()`, or `@Body()`. TypeScript provides types for objects either through interfaces or classes. Since interfaces are only utilized during the compilation step, they disappear when the code is transpiled to JavaScript, and should be preferred whenever possible. However, classes allow you to easily create multiple instances of its structure, utilize inheritance, and support decorators where interfaces do not. Classes are compiled to objects in JavaScript and bloat the application, and should not be used as interfaces. For validation, classes are required as interfaces do not appear during runtime.

The validating structure by practice is a class called a data transfer object (DTO). Each property must have at least one decorator to be considered within the validation schema. TypeScript cannot discern types from the decorators, thus each property must also be typed properly as seen with the age property in Figure 21. The decorators available can be the basic types, such as number, string, boolean or contextual ones such as emails, phone numbers, JWTs, urls, decimals, containing certain strings, number minimums and maximums, and a few dozen others [22].

```ts
export class CreateUserDto {
  @IsString()
  password: string;

  @IsEmail()
  email: string;

  @IsDate()
  @IsOptional()
  age?: Date;
}

@Controller('user')
export class UserController {
  constructor(private readonly userService: UserService) {}

  @Post()
  create(@Body() { age, email, password }: CreateUserDto) {
    return this.userService.create(email, password, age);
  }
}
```

Figure 21: A destructured data transfer object (DTO) for validating an endpoint.

5.9.8 File Upload

Uploading files utilizes the native HTML form element and the backend library Multer to parse the file stream [23]. The form element supports three encoding types it can use to package and transmit form data to the server but only one of which supports files.

Form Encoding	Explanation
`application/x-www-form-urlencoded`	The default form encoding that generates a query string by using the name of form element as the key and the answers as the value. This adds one byte per field separation with the & symbol and is more efficient with text, however adds 3x the bytes per non-textual character that needs to be encoded. Simply does not work with files.
`multipart/form-data`	Each value is blocked with an agent-defined delimiter. There is some overhead calculating the delimiter, and uses slightly more

	space, but supports file uploads.
`text/plain`	Intended on being a human-readable format that ambiguously formats text and is thus not suitable for computers.

The native form element also supports submitting a POST request to a defined URL without any special libraries. This functionality is likely to be overridden to support client-side validation libraries. JavaScript provides a FormData interface for manually constructing the correct multipart/form-data encoding which may be easier for incorporating with state management within React.

On the backend, the Multer library will stream the files it finds, validate if they meet restrictions, then save them or remove them. Multer has support for custom storage options, such as Amazon S3, on top of saving directly to the disk. Placing restrictions on the uploaded files is important to limit the possibility of an adversary attempting to commit a distributed denial of service (DDOS) attack by starving the server of memory, space, or bandwidth. Uploaded files are inherently insecure and should also be written to a system that will not be potentially executing them.

Multer, acting as an interceptor for the associated route seen in Figure 22, will pass any form data to the controller for validation. Once the controller is passed control, the files are either uploaded or multer has thrown an error. Some instances of multer require single files, or situationally allow any number of files, and may pass control without error.

```
@Post("reduced-lunch-form")
@UseInterceptors(FileInterceptor('reduced-lunch-form'))
uploadReducedLunchForm(@UploadedFile() file) {
  return this.userService.addFile({ name: 'reduced-lunch-form', file })
}
```

Figure 22: FileInterceptor for a single file invoking Multer.

5.9.9 Database

Object-relational mapping (ORM) is the conversion of one data structure to another using an object-oriented approach. Entities within a database are often large tables less

appropriate for managing nested structures and relations. Relations in SQL are typically joined by mixing columns and making the hierarchy of data less understood and harder to manipulate from its results. ORMs seek to turn entities into objects that hold their attributes as properties with nested relation objects more compatible with the data structures found in programming languages.

However, the use of ORMs is contentious as the reason SQL exists is to provide a friendly apparatus for developers to interact with databases. This abstraction often leads to poorer understandings of how SQL works and the database the developers are working with. This abstraction can be useful for tediously simple queries though, and most ORMs provide raw SQL querying as a fallback. Knex.js for example, considers itself more of a query builder, using familiar SQL verbs as methods that build SQL queries without occluding how SQL operates [24]. MikroORM focuses on being a performant entity by utilizing the concept of Unit of Work, or an implicit wrapping of all operations within a transaction, entity building and management within the backend, migration support, a powerful query builder, while building a strong bond with TypeScript not commonly found in other ORMs.

Configuration is done for the root of the application either in a standalone configuration file or in the root `AppModule`. The latter approach is ideal for our application as it allows us to use a configuration system to inject environment variables across the backend in one location. There are a few important utility features that MikroORM provides: migrations, synchronization, entity glob patterns, and schema dropping. During the configuration of MikroORM, a glob pattern is set such as `"/src/**/*.entity{.js,.ts}"` to determine the location of all entities within the application.

5.9.9.1 Active Record vs Data Mapper Patterns

Describing entities within MikroORM is done with the Data Mapper pattern. In other frameworks or ORMs, the use of the Active Record pattern may be more prevalent. Methods are called on the entities themselves in the Active Record pattern, whereas Data Mapper relies on a repository to be retrieved for each entity before utilizing it. Entities within a Data Mapper pattern are merely buckets of data and need to be passed to the repository to be acted upon. Data Mapper is more popular due to its explicitivity and careful separation of powers. With Active Record, it is possible to manipulate entities through their inherited static methods across the entire application which may be

appealing, but is a poor design practice. Similarly, Active Record is much harder to write tests for, as the repository provided by Data Mapper can be easily mocked.

5.9.9.2 Defining Entities

Entities are described in MikroORM using classes and decorators much like the NestJS. An example of a project entity without relations can be seen in Figure 23. It utilizes the class-level `@Entity()` decorator to mark the class as an entity and optionally allows it to be renamed, e.g. an entity name could be singular within the application, but the database table name can be plural. The entity then requires a `@PrimaryKey()` decorator to describe what the primary key is within the database. The type of data can be anything the underlying database supports as a primary key, however it defaults to an auto-incrementing integer sufficient for the website. Standard, non-relational, properties are then referenced using the `@Property()` decorator.

These decorators have an optional level of explicitness that can be described with an options parameter. Meaning, using standard TypeScript and describing the property as optional with the question mark before the number type would be properly denoted as an optional (nullable) field by MikroORM. However, relations do not always support this level of implicitness, nor do certain types such as enumerables. The default value can also be set within the class property itself or using the decorator.

```typescript
@Entity()
export class Project {
  constructor(name: string, description: string, hours?: number) {
    this.name = name;
    this.description = description;
    this.hours = hours || 0;
  }

  @PrimaryKey()
  id!: number;

  @Property()
  name!: string;

  @Property({ nullable: true })
  description?: string;

  @Property({ type: 'smallint' })
  hours: number = 0;
}
```

Figure 23: Data Mapper pattern using a repository to manage an entity.

Defining relations in MikroORM can be done using specialized decorators for all of the common types, described in the table in Figure 24. However describing a relationship side with many entities is not done with typical arrays, Instead, MikroORM describes a collection interface for dynamically loading, counting, and inserting entities. Relation definitions between the user and account entities can be seen in Figure 25.

Relationship Decorator	**Description**
`@OneToOne()`	A single entity can reference another single entity. This can be one-way or two-way, allowing the relationship to be traversed from either entity. The owner must be specified, either by making it one way, or with the owner option. Users will only have one account, and accounts will only have one primary user.
`@ManyToMany()`	Many of an entity can have many of another. Requires a collection. This creates a join table that can be renamed, if necessary.

	If roles were to be described in the database, a user could have many roles, and each role can have many users.
`@OneToMany()`	An entity that relates to many of another kind of entity. Requires a collection. Accounts can have many users.
`@ManyToOne()`	Many of the entities on this side refer to another entity. Does not require a collection. Many users will have one account.

Figure 24: Table of entity relationship decorators.

```
// Account side.
@OneToOne({ cascade: [Cascade.ALL], orphanRemoval: true })
primary_user: User;

@OneToMany({
  entity: () => User,
  mappedBy: 'account',
  orphanRemoval: true,
  cascade: [Cascade.ALL],
})
users = new Collection<User>(this);

// User side.
@Property({ persist: false })
account_id: number;

@ManyToOne()
account: Account;
```

Figure 25: Example relationship models of accounts having one primary user and many users, and users having one account. The account_id is the join column. Exposing the join column is useful for cascading inserts using entity ids.

5.9.9.3 Migrations

To speed up development, MikroORM provides a method of updating the database schema from the definitions within the entity files. This is done using the command line by either forcing a schema update, or by generating migration files. Not every database data type or feature is supported implicitly by MikroORM, but there are methods of describing custom data types within MikroORM. One has already been created for our project: arrays of enumerables for describing user roles, shown in Figure 26. A migration is then created and the changes entered into these files, then the migration command is run and the database will have the changes persisted. Each migration has a timestamp that is stored in the database so MikroORM knows not to run it again.

Migrations utilize the concept of "up" and "down" directions for each file. This allows changes to be undone in the event they need to be reverted. These files are committed inside the repository so other developers have the most up-to-date schema on their development databases as well. This feature will not likely be utilized much if at all by OMC during production as it is possible to run migrations programmatically, meaning each can be coded into the startup sequence of the application to make it seamless for the clients if changes ever need to be made to their database. Documentation on how to generate new migrations, making backups, and running migrations will be provided to the clients as well.

```
export class Migration20200415034728 extends Migration {
  async up(): Promise<void> {
    this.addSql(
"create type user_roles_enum as enum ('Admin', 'Educato
r', 'Volunteer', 'Verified', 'Unverified');"
);
  }

  async down(): Promise<void> {
    this.addSql('drop type user_roles_enum;');
  }
}
```

Figure 26: Migration for creating the user roles enumerable.

5.9.10 Authentication

Nest.js provides guidance on how authentication strategies can be constructed and wired up to guards, but the implementation is ultimately a tailored experience. The Passport library provides a modular authentication structure that meshes well with Nest.js and allows for expansion into social login with OAuth2 in the future if the website were further developed [25].

Each passport strategy involves creating a class that extends the strategy package installed. Two such strategies our application will need are `passport-local` and `passport-jwt`. The local strategy is quite simple, passport is expecting a username or email field and password field in a POST request body that it will pass to the validate method shown in Figure 27.

```
@Injectable()
export class LocalStrategy extends PassportStrategy(Strategy) {
  constructor(private readonly authService: AuthService) {
    super();
  }

  async validate(email: string, password: string): Promise<any> {
    // Bcrypt will rehash their password using the stored salt.
    // If the hashes match, the authentication is valid.
    const user = await this.authService.validateUser(email, password);

    if (!user) {
      throw new UnauthorizedException();
    }

    return user;
  }
}
```

Figure 27: Local passport strategy for checking login validity.

Invoking a strategy is done by utilizing a guard. Nest.js comes with guards that can invoke passport strategies by name by including them above controller routes, shown in Figure 28. If authentication is successful, they will append the user entity retrieved during validation to the request object and will be available to the rest of the route if necessary.

Custom guards will likely be made extending the original AuthGuard to incorporate permissions and better error handling.

```
@UseGuards(AuthGuard('jwt'))
@Get('profile')
getProfile(@Usr() user: User) {
  return user;
}
```

Figure 28: A protected controller method and a decorator returning the user.

5.9.11 Queues

Once approved, the AWS Simple Email Service (SES) will provide a quota that describes the total number of emails that can be sent daily and secondly. The SES endpoint for sending emails is rate-limited by the number of emails sent, not the total number of calls to the method. Even if the limits are high enough for OMC to never saturate them, website performance will suffer if a background task is sending a high enough volume of emails in a short enough time frame.

Queues provide the ability to smoothen out the number of asynchronous requests fed to it by specifying concurrency limits and other rate limiting parameters to stay in line with the quotas described by external APIs such as SES. Emails can be sent in a separate process, improving the user experience by not requiring they wait for registrations and password reset emails to be sent before approving their request; those jobs are simply added to a queue. The difference between developing a queue from scratch and using a popular library lies in the need for persistence, failure retrying, and prioritization.

Two popular libraries that go hand-in-hand are p-queue and p-retry. The former provides a queue class that accepts rate limiting parameters and has methods for adding jobs. Jobs can be added with priorities, useful for making administrative emails send faster than notification emails. To handle cases where the API returns an error code, the second library is used by wrapping the asynchronous call and describing the retrying pattern. An example of which is an exponential backoff pattern with a maximum number of tries, so it retries on an exponentially increasing interval and gives up when it has tried too many times. These two provide an adequate queueing system, however these libraries consume the job and the only status updates provided are through asking the queue class if there

are any pending or active jobs. Figure 29 shows an example implementation that the email service could utilize. The example queue limits only 5 concurrent jobs active simultaneously, with a cap of 50 jobs completed in a second and the second method uses p-retry on the default exponential increase up to a maximum of 5 retries.

```ts
import PQueue from 'p-queue';
import PRetry from 'p-retry';

export class EmailService {
  private readonly queue = new PQueue({
    concurrency: 5,
    interval: 1000,
    intervalCap: 50,
  });

  public async email() {
    await this.queue.add(() => aws.sendEmail());

    // Or with retrying.
    await this.queue.add(
      PRetry(aws.sendEmail(), { retries: 5 })
    );
  }
}
```

Figure 29: Example email service class with p-queue and p-retry.

Bull, a popular Redis-backed library for creating job queues, is more thorough compared to p-queue and p-retry [26]. Bull stores jobs in Redis, allowing for them to continue even if the server crashes or restarts. A created job returns an id that can be sent to the client so the frontend can poll the status of the job to let the user know the progress of their job. Each Bull queue can have multiple processors that map to different types of jobs with event listeners so the status of events can be logged or recorded. Since Bull has a NestJS module, it can be described modularly and injected in other services that need to utilize the queue, depicted in Figure 30.

```ts
export class UserService {
  constructor(
    @InjectQueue('email') private readonly queue: Queue,
    private readonly emailService: EmailService
  ) {}

  public async forgotPassword(id: number) {
    const template = await this.emailService.getForgotTemplate(id);
    const job = await this.queue.add({ priority: 2 }, template);
  }
}

@Processor('email')
export class EmailQueue {
  @Process()
  async sendEmail(job: Job<EmailTemplate>) {
    await aws.sendEmail(job.data.template);
  }
}
```

Figure 30: Bull queue being used in a service and its processor.

We are planning on utilizing Bull as it has recommended libraries for NestJS integration and has more features that may become useful if the website ends up utilizing more APIs, for example, Google Calendar. Redis can be configured to not persist data to the disk if persistence is not wanted, and will consume few resources.

5.9.12 Websockets

The attendance page for admins will need to incorporate a means of refreshing the page data ideally without requiring the admin to do so manually. It will be common that new parents to events are not registered to the website and can be told to quickly fill out the website's registration form on the spot. The attendance roster should automatically fetch users registered to it by subscribing to changes in registrations for that event. Users who are not registered will also be searchable at a glance to add them to the attendance page, possibly with a list of users who had just recently joined the website for quick reference.

The implementation utilizes the publisher-subscribe pattern. This involves a consumer that will retrieve the data from the publisher by first telling the publisher it is interested in updates. The publisher is not aware of a specific device to send this information to, only those who subscribe to the information. In the past this functionality was commonly done

with various lower-level network protocols or through long-polling, a practice of the client simply making requests to the backend repeatedly. Long-polling is easy to develop, however it utilizes more backend resources as the server will likely send the same response many times, often by a more expensive lookup procedure. Websockets, which will be implemented by the Socket.io library, are a communication protocol on persistent TCP connections for sending bi-directional messages [27].

Nest.js provides Websocket functionality through a specialized provider file called a gateway. They operate much like controllers in the way that they use decorators to bind to some identifier that a router will use to handle the proper communication. However, gateways are pushed and not directly called. By default, the Websocket connections are done on the same port as the server, but it can be defined in the class-level @WebsocketGateway() decorator if a secondary port is required. Figure 31 shows how a simple gateway with only the welcome messages would be made.

```
@WebSocketGateway({ transport: ['websocket'] })
export class EventsGateway implements OnGatewayConnection, OnGatwayDisconnect {
  @WebSocketServer()
  public readonly server: Server;

  handleConnection(client) {
    console.log('Client Connected');
    client.emit('connection', 'Connection successful.');
  }

  handleDisconnect(client) {
    console.log('Client Disconnected')
  }
}
```

Figure 31: Example websocket gateway in Nest.js

The gateway file itself is mostly used for responding to client-emitted messages. A special authentication procedure will also need to take place over the websockets to ensure the initial connection is properly authenticated, but is compatible with guards, middleware, and interceptors. However, once this gateway is injected into another provider or controller, the server can be used to send emissions of its own, likely after the creation of certain entities.

To accomplish this on the server, the client-side version of the Socket.io library is loaded on the frontend and can be used to subscribe to events. During the lifecycle hooks of the attendance page and before the page is mounted the Socket.io library can be told to subscribe to an event with a specific id, then when the server receives registration updates it will send the updated event data to those subscribed to that event id using the `emit()` method. Although the number of pages that will likely utilize websockets is small, it will be important to ensure when the page is being destroyed that the websocket connection is also destroyed. The websocket will already be intended for long-duration use, however if multiple connections are concurrently established it can worsen application performance.

5.10 Frontend

Frontend development has historically been tightly coupled with some backend language in a monolithic structure. PHP, the most popular language that does this, fundamentally takes a HTML document and injects the PHP language into it with special tags. Then when the user requests a page, the PHP server will render the page after having retrieved and manipulated the data, called server-side rendering (SSR). Server-side rendering does not require that the backend be intermingled with the frontend, however, only that it has processes that compile and retrieve data on the server before hydrating the page and sending it to the user fully-formed.

Every time a link is clicked on a standard website the current page is discarded and replaced with the newly retrieved one. This is less efficient in many aspects because most websites have parts that maintain a homogenous experience across the website, such as authentication, the header, and the footer. A single-page application takes over the routing experience and retrieves only the pieces that need to be updated as necessary. This makes the website operate more closely to a native application than a standard website, hence the name. However, the code necessary to perform this routing adds overhead to the application especially when used in conjunction with JavaScript libraries which describe website content primarily through JavaScript. On slower connections users could be waiting in front of a blank page until the heftier JavaScript loads and takes control. Server-side rendering can thus be used once more to pre-render the first page load on the server before the SPA takes control of the routing, providing the best of both technologies. We feel that it is important to be as performant as possible for a demographic which includes older technology in areas with poor mobile connections while not sacrificing the improved developer experience of SPAs.

To make developing the frontend as approachable as possible we had set our sights on a component-based library or framework. The React library popularized web components, or the ability to make encapsulated, reusable, and repeatable blocks of code within the frontend. To compare, object-oriented programming requires that developers understand how each object works to utilize it, whereas component-based programming does not. When joined with a component UI library, many of the website features that were previously tedious to scaffold can be made in a few lines of code by using or extending pre-existing components. Paired with a UI design tool such as Figma, we can quickly develop the basic building blocks of the website using concise and modular code. With that in mind, we have thoroughly researched Angular, React, and Vue as they are the most prevalent component-based libraries with a sufficiently large community backing. However, upon initial review we found a community exodus away from Angular due to the library being completely redesigned from its AngularJS days and we have since not considered using it.

5.10.1 React vs. Vue

React is an open-source, declarative component-based library for building user interfaces pioneered by Facebook [28]. It popularized the idea of programming with components in JavaScript and introduced a more efficient means of updating the DOM thanks to its own Virtual DOM. Before React, jQuery and vanilla JavaScript would need to target DOM elements for manipulation by scanning through the tree of nodes, a reasonably expensive procedure and not as developer friendly. Conversely, React takes control of the entire structure of the document and stores each node in memory (virtually) and algorithmically compares it to the current DOM whenever an update occurs. Being able to write code without having to worry about how it's controlled made it an immediate developer favorite. Businesses favor React as well as it is being backed by Facebook, a large corporate entity.

React code is written in JSX, a JavaScript and HTML hybridization that writes all code in JavaScript. JSX is created in either functional components, shown by example in Figure 32, or class components. JSX is returned from components to be rendered by React into nested nodes in the DOM. To manage data, lifecycle methods are called at various points of the creation, destruction, and updating processes for class components, and similar means exist using hooks within functional components. React also introduces props, or properties that can be passed to child components as a means of trickling data and methods down the tree. To simplify and keep React performant, React only uses one-way

data binding, meaning child components cannot directly mutate state, but functions can be passed to children that perform mutations in parent components.

```jsx
const BlogPost = (props) => {
  return (
    <div>
      <span className="title">{props.title}</span>
      <p>{props.summary}</p>
      {props.links.map((link) => <Link key={link.id} to={link.to} text={link.text} />)}
    </div>
  );
};

export const BlogPost;
```

Figure 32: Stateless React functional blog post component example written in JSX.

The Vue.js framework is the newest of the trio, nonetheless, it has gained considerable popularity as the second most popular component framework even though it lacks a corporate backer like Google's Angular or Facebook's React. Vue was created by taking the best parts of Angular and React and consolidating them into a more incrementally adoptable format with an easier learning curve without making any compromises to performance. While JSX requires a compilation step, Vue can be used directly in the browser which allows for legacy applications to try out Vue without needing to completely overhaul the application. Though, a complete Vue application does end up requiring a compilation step.

Building components in Vue can either be done by directly creating a new component instance with `Vue.component({})` and providing a template string similar to React's class components, but when a compiler is available Vue is generally written using single-file components. Each SFC has three sections shown in Figure 33: the HTML, the JavaScript, and the CSS. Other templating engines (or even JSX, though not common), TypeScript, and a different CSS processor may be used. This separation of concerns is a major contributing factor to Vue's rise in popularity. Instead of using JavaScript to manipulate components and the DOM, they are manipulated with directives, a feature borrowed from Angular.

```vue
<template>
  <div>
    <span class="title">{{ props.title }}</span>
    <p>{{ props.summary }}</p>
    <Link v-for="link in links" :key="link.id" :to="link.to" :text="link.text" />
  </div>
</template>

<script>
export default {
  components: {
    Link,
  },
  props: ['title', 'summary', 'links']
}
</script>

<style scoped>
.title {
  font-weight: bold;
}
</style>
```

Figure 33: A Vue single-file component adaption of the React functional component in Figure 32 with an example of scoped styling.

Ultimately, Vue was chosen as our component framework instead of React. When comparing the use of hooks or lifecycle events against those of Vue, there are a lot more performance considerations that must be rationalized with React. A few prototypes were made to compare the data flow necessary for authentication in both, and while Vue was rather straightforward the React documentation was subpar and not easy to conceptualize and we wasted a lot of time even though the processes were directly translatable. Since the website will have an educational component to it, we feel that Vue will make more sense as while its user base is smaller, the packages are of a similar or higher caliber and with notably better documentation.

5.10.2 Figma

Figma is a browser-based design tool used for creating user interfaces [29]. It provides users with the ability to create and store interface mockups quickly and easily with no external software or licensing necessary. Once they are logged in, users can create a new project. This new project will begin as a blank page that users can add new components, assets, and pages to the UI. Any changes made, whether they be to the font, background

color, or any aspect of the web page, will be recorded in the .css file. Users can access this file at any time to view the code for their design choices, and can take this code when they are finished so that they can implement their design in their website.

Another excellent feature of Figma is its included support for group projects. Much like Google drive, Figma allows users to share a project with as many group members as they would like. This makes it very easy for users to design a UI that their entire group is happy with, and all group members can go in and change anything they would like whenever they would like. The interface can be seen in Figure 34.

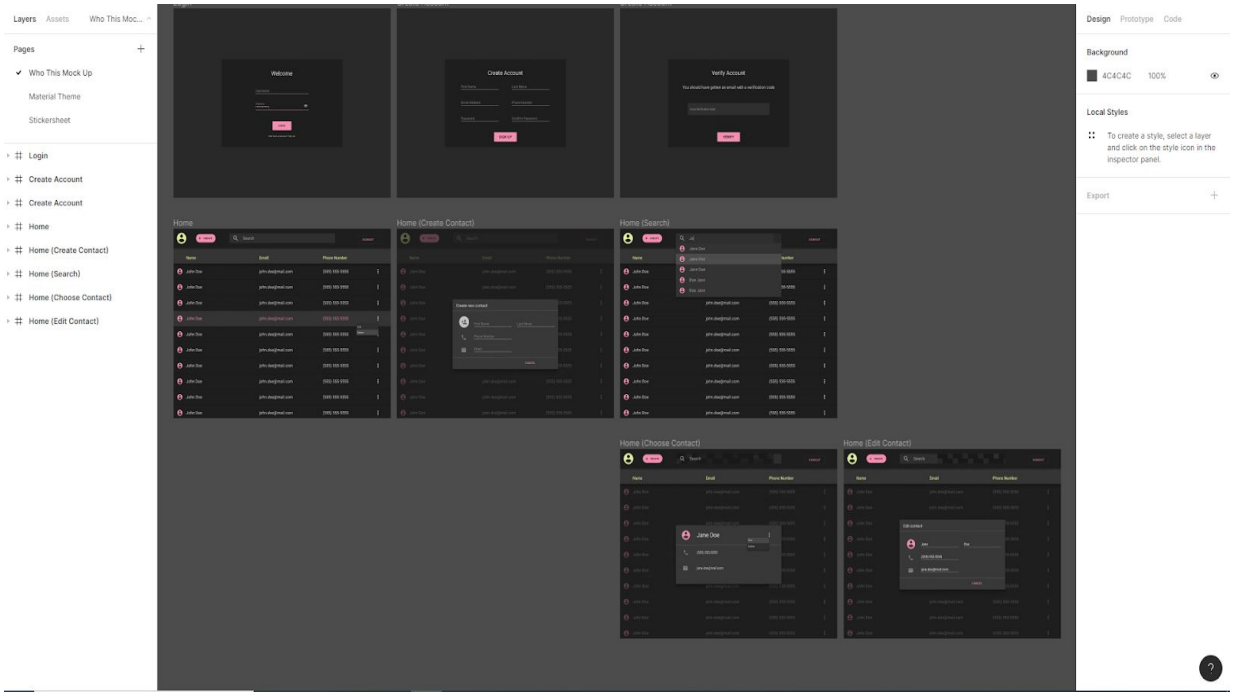


Figure 34: A user interface created with Figma

On the left side of the page, you can see each layer of the created interface. These layers can be used as either new pages for the website being created, or simply pop-ups that occur on other pages.

On the right side, you can see the color of the background for the website. If a layer is selected, then the styles for that layer can be viewed on the right side. Each layer contains several different components, and users can cycle through each of these components to view the accompanying css code. This makes it very simple for users to design a UI they like, and use the css code for their website, instead of trial and error with a .css file and constant refreshing of the webpage.

Figma also has a prototype tab on the right side of the screen as well. This tab allows users to connect certain layers that are meant to work together in a certain order. For example: in the example above, the login screen is connected to the home page, and the home page is connected to the create, edit, search, and delete contact screens. This is shown in Figure 35.

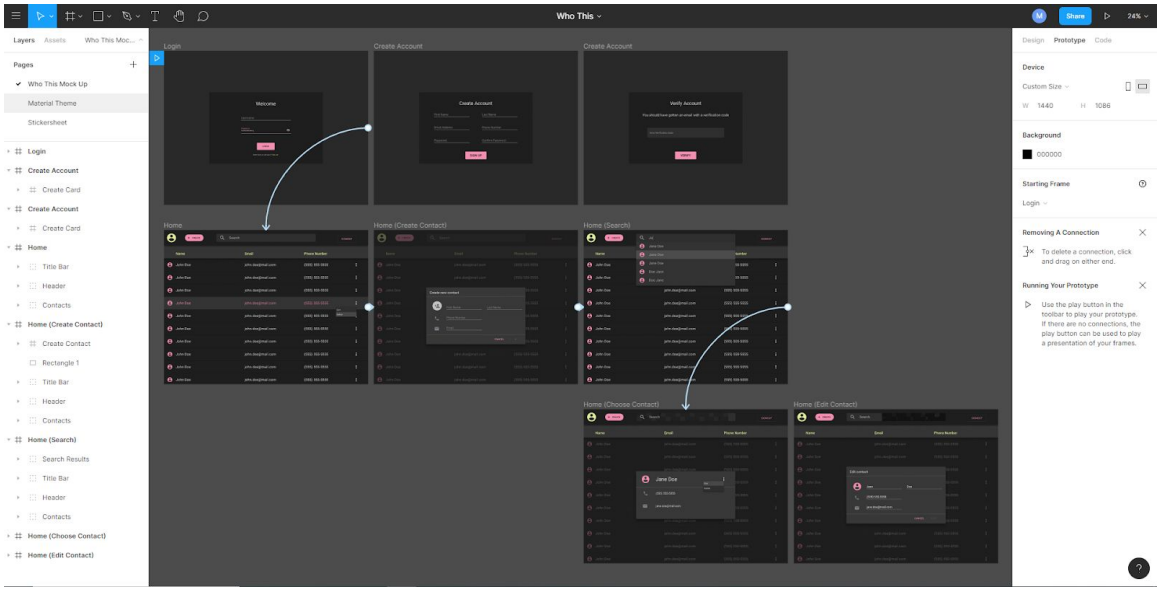


Figure 35: Connecting UI layers using Figma

As users connect layers together, Figma draws arrows between each layer so that connections are easily visible at a glance. Once users have connected the layers of their interface, they are able to view a quick preview of how the layers will flow together. In the top right corner of the page, there is a play button, located next to the share button. This play button allows users to see how the layers will proceed, from the opening layer that they have set, to the last layer.

Figma also allows users to set color themes and fonts for the entire page without having to select them on each layer. Color themes can be set and selected for any part of the UI, allowing the UI to have the same exact color scheme without needing to remember the color selection or copy and pasting the color values from somewhere else multiple times. Fonts can be set for the entire website, with different sizes and stroke-thicknesses being specified for different parts. Headings can have different sizes depending on their importance, including making some bold, underlined, or italicized.

Users also have the ability to create specific designs for any buttons or links that are present in their UI. Figure 36 shows the settings for one set of buttons on the UI being created. The user can set different styles for the same button depending on specific event types. The button will change when it is hovered over, clicked on, or dragged. This design can be linked to the button so that if the button is used in multiple places, it will always have these styles.

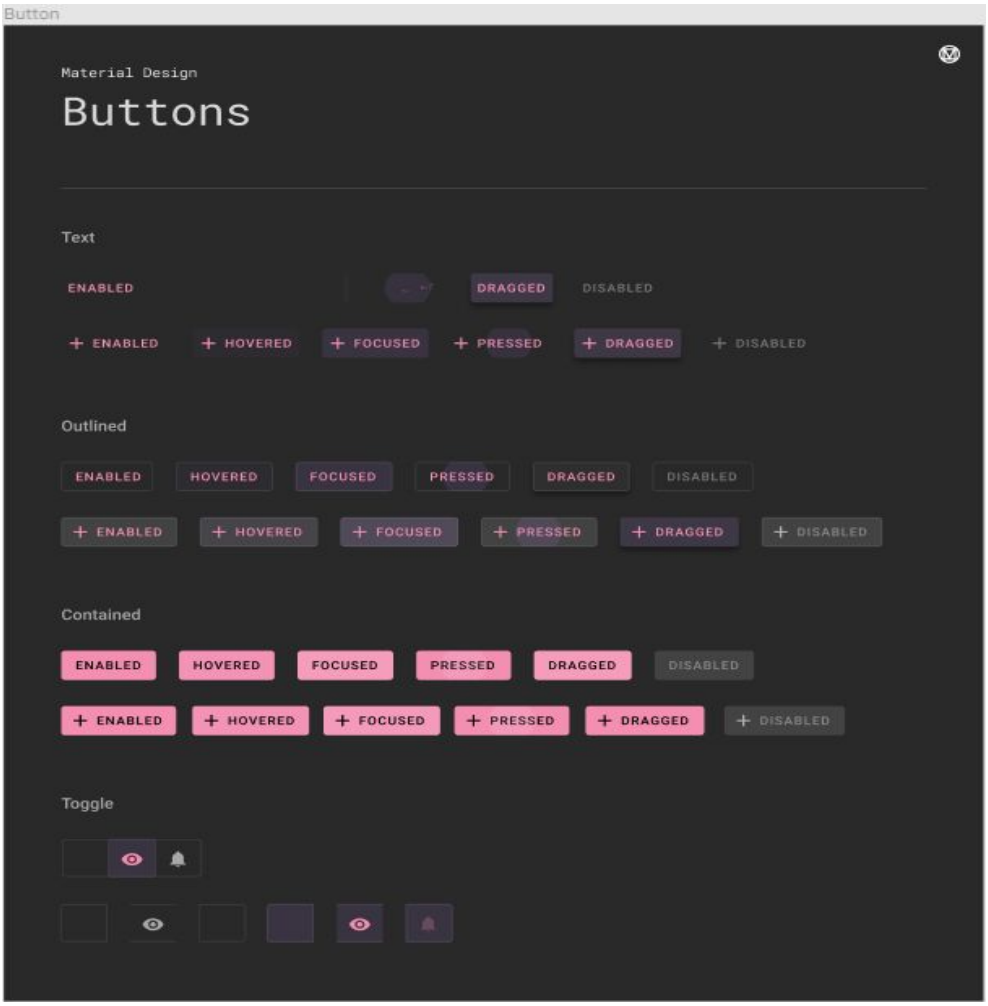


Figure 36: Custom button designs for UI

5.10.3 Component Template

The first tag within a SFC is the template tag which contains spec-compliant HTML for the component. Template tags can be nested, often used to conditionally render parts of the component without leaving behind an empty element. The template tag is used to convert HTML into Vue nodes, and as a result there can only be one top-level element within the root template tag. A div may be used as a top-level element if it is not possible to structure the HTML around this limitation.

The HTML within the template uses directives to manipulate the DOM based on the state, or by binding properties to components. Directives are custom attributes that enhance the behavior of the website by abstracting away common DOM manipulations. The default directives are transcribed from their explanation on the Vue API documentation in a table on Figure 37 [30]. Custom directives can be made using Vue as well, for instance to create an intersection observer that waits for the element to be in view before executing some action.

Directive	**Function**
`v-text`	An alternative to the moustache braces to insert text into an element.
`v-html`	Renders dynamic HTML from a string within the element. This is dangerous as dynamically rendering arbitrary HTML is a XSS vulnerability. Scoped styles will not apply to HTML inserted this way.
`v-show`	Toggles the display property of an element based on the truthfulness of the value.

`v-if`
`v-else`
`v-else-if`
	Conditionally renders elements based on the truthfulness of the value. The else and else-if directives can be used on any same-level subsequent element(s) or template tags.

`v-for`	Loops through arrays, objects, numbers, or iterables. It uses the “alias in expression” syntax, where alias is the current element within the iteration. When used, the key property must also be specified on the rendered element. The key should be a unique value to the item being rendered, such as

	an id from the database, in order to give Vue a hint on if it needs to re-render that element when the iterated structure is modified. If there is no such unique identifier, the alias can be wrapped to include the second parameter, a counting iterator and used in its stead: “(alias, i) in expression”.
`v-on`	An event binder. For instance v-on:click would call the function provided by this directive on every click. It has a few modifiers, such as `v-on:click.stop` and `v-on:click.prevent` for `stopPropagation()` and `preventDefault()` respectively. The “`@`” symbol can also be used as a shortcut for v-on, e.g. `@click="myFunction"`.
`v-bind`	Dynamically binds one or more attributes, such as an object of attributes, to the element. For example, `v-bind:src="imgSrc"` would bind the src attribute to the imgSrc value. Using v-bind in this way is verbose and not common, the v-bind can be omitted altogether for a single attribute, `:src="imgSrc"`. Using v-bind by itself is typically only used for binding multiple attributes at once. If the attribute name is unknown, brackets can surround the property, e.g. `:[key]="someValue"`.
`v-model`	Creates two-way binding on a variable. This can only be used on the input, select, and textarea form elements as well as other components. With components, the initial value can be bound to wherever it needs to be read in the component as a default value, and updates can emit the input back to the parent component.
`v-slot`	Slots are a special way of denoting areas of components where custom HTML can be injected. Many component libraries have a default styling, but allow slots to override them if custom structures are needed. This is similar to the concept to children in React. If a component has a single slot, it may be defined using a `<slot/>` tag. It will render as a default value when no children are provided. However, when multiple slots are needed, slots need to be named, e.g. `<slot name="header">`, and the parent component will use v-slot on a template tag to denote which slot to utilize. Slots may also bind data, called scoped slots, `<slot name="header" :user="user">`, which would be accessed as `<template v-slot:header="user">`, allowing access to the user object.
`v-pre`	An element with the pre directive will not be compiled or any of its children, allowing the display of the HTML, component calls, and mustache tags.

`v-cloak`	Hides the element until compilation finishes.
`v-once`	Instructs Vue to only ever render the component once. On subsequent re-renders, it is treated statically and skipped.

Figure 37: Vue.js directives and their function within a component template.

5.10.4 Component Options API

The Options API that is used for describing components is a layer of abstraction for describing and manipulating data. TypeScript will be used on the frontend, however, the Options API is not fully compatible. Property and template type checking does not exist due to the layers of abstraction around the "this" keyword. In Vue 3.0, which is currently in beta, the library provides a Composition API as an optional replacement for the Options API which has much better TypeScript support. Libraries do exist that augment or decorate the Options API, such as a class-based component approach with decorators, however, they would make the code more difficult to follow for future maintainers as code would no longer match the official documentation.

The Options API is used by exporting a default object, or by explicitly creating a Vue component with an object in the constructor. The data properties and lifecycle methods can be described within the object. The common data properties and the components property are listed in the table in Figure 38, and an example usage of the Options API in Figure 39. To reference other properties or methods from within a function inside the Options API, the "this" context is used as it contains the hoisted data and computed properties, props, and methods.

Data Property	**Function**
`components`	An object of comma-separated imported components that are used in the template. The component must be imported at the top of the script tag before they can be used.
`data`	A function that returns a plain object containing the state values for the component. For them to be reactive, they must have an explicitly defined default value. Null is an acceptable default value. Properties cannot be added to the data list dynamically, however objects and arrays can have properties dynamically added by either replacing the object entirely with `Object.assign()` or using `Vue.set()`. This is a reactivity

	caveat to Vue that is going away in Vue 3.
`props`	An object or hash array of the properties the component expects. For type checking, the object syntax allows defining the expected type through its constructor (e.g. String, Number, Object, Function), an optional default value, if the prop is required, as well as a validation function. TypeScript is most useful in this situation as you cannot define the properties of prop objects here without writing lengthy custom validators. Props can be read directly, however they cannot be directly mutated such as with the v-model directive. An emitter can be used to send an input event to the property, allowing the value to be updated in the parent component if necessary.
`propsData`	An override to providing prop data when a component is created. This is used for unit testing components.
`computed`	Computed values are defined by a property and a function that returns a value that is expected to change with the reactive updates of the values within the property. The value of a computed property is cached and only updated if the values it depends on are changed.
`methods`	The methods property allow for functions to be defined within a component. They may perform reactive updates, be called in template tags, and called in lifecycle updates. However, they should not be arrow functions as context of the “this” keyword, used for accessing data and other methods, would be lost.
`watch`	Similar to computed properties, watchers perform an action when a property is reactively changed. Computed properties are cached, and should be used where possible, however watchers are useful for expensive reactive actions, such as a debounced search box calling an API method. Watchers also are generally meant to be used to update values.

Figure 38: Vue data properties as well as the component property for imported components.

Lifecycle hooks are special methods also available within the Options API. They are called at various stages of a component’s creation, updating processes, and deletion. Access to the other methods, watchers, props, computed properties, and state is not ready until the created hook is called. Similarly, DOM manipulations will not work until the

mounted hook is called, as it is when the interface is created. Once mounted, reactivity changes that require the interface to be rerendered will call beforeUpdate and finish with the updated hook. These hooks will not be used often in our application as there are other hooks provided by Nuxt.js, however, in a vanilla Vue.js application the created hook is most commonly used for fetching information. If fetching data were done later in the lifecycle, the component would inefficiently render twice on every first load.

1. BeforeCreate
2. Created
3. BeforeMount
4. Mounted
5. BeforeUpdate
6. Updated
7. BeforeDestroy
8. Destroyed

```
<script>
export default {
  components: {
    Link,
    Header
  },
  props: {
    title: {
      type: String,
      required: true
    }
  },
  data() {
    return {
      firstName: 'Jane',
      lastName: 'Doe',
      search: '',
      records: []
    }
  },
  computed: {
    fullName() {
      return `${this.firstName} ${this.lastName}`
    }
  },
  watch: {
    search() {
      this.debouncedGetRecords = _.debounce(this.getRecords, 500)
    }
  },
  methods: {
    getRecords() {
      const data = await this.$axios.$get('/records',
        { query: { search: this.search } }
      )

      this.records = data
    }
  }
}
</script>
```

Figure 39: Example component Options API usage. When the search value is changed, the watch property will call the `getRecords()` method at most once every 500 milliseconds.

5.10.5 Component Styling

Vue provides a means to declare styles by scoping them to the component or globally within a single-file component. Since SFC's require a compilation step, CSS can be parsed by the autoprefixer library and minified by the compiler without worrying about browser support or performance optimizations while writing styles. When dynamic styles are required, Vue has directives for dynamically applying classes as well as inline styles.

Writing styles within components is done using the style tag at the root level of the file. Two attributes are supported by the style tag: global, meaning any styles within the tag apply throughout the application and lang, allowing for a different CSS processor to be used, shown in Figure 40. In the figure, SASS is utilized to create a reusable mixin that would style every other element with the post class differently. Since the block has the scoped attribute, the styles will not be reusable elsewhere, eliminating the need for complex naming patterns to avoid identifier collisions.

```
<style scoped lang="scss">
@mixin bold-solid-shadow($color) {
  border: 2px solid $color;
  box-shadow: -9px 8px 0 $color;
}

.blog {
  .post {
    @include bold-solid-shadow(#f0f0f0);
  }
  .post:nth-child(2) {
    @include bold-solid-shadow(#151515);
  }
}
</style>
```

Figure 40: Vue SFC style block scoped to the component and written in SASS.

To avoid collisions, components include a data attribute on all elements that have class or id identifiers and the scoped styles will include the attribute identifier when selecting the element. A screenshot of a compiled Vue style being applied to a DOM element is shown

in Figure 41. It can be faintly seen in the screenshot that the styles for the component are extracted into a css file, allowing the browser to cache the file for subsequent requests. Vue by default extracts CSS to style tags in the header. CSS extraction to files is provided by the Nuxt.js library, discussed in the frontend framework section.

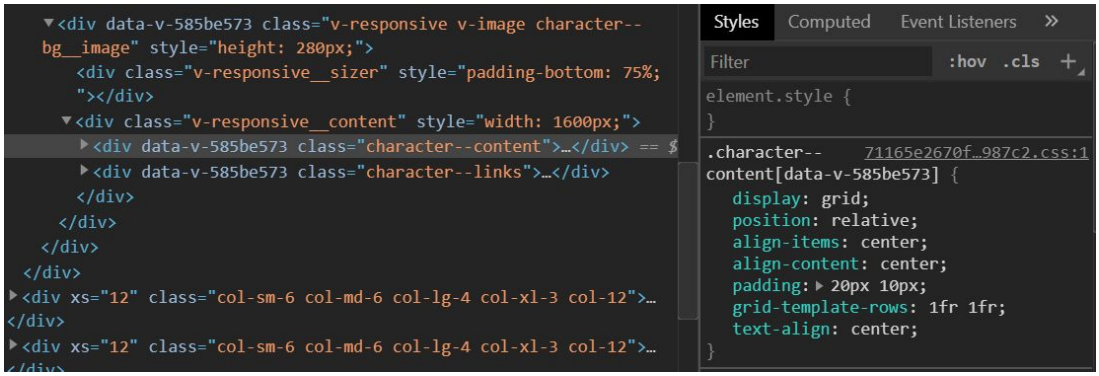


Figure 41: Chrome DevTools inspection of a production Vue website with scoped CSS.

When the presence of a class name needs to be toggled by some truthy value, the v-bind:class directive can be used by providing as its value an object where the property names are the classes and the values are the conditions. So long as the conditions are reactive, the class or classes will be added or removed as necessary. When inline styles are necessary, such as if colors are part of data retrieved with a page or component, the v-bind:style directive can be used. It operates similarly for classes; camel-case (or quoted kebab-case) properties representing the css properties are the keys of objects and the values are the string representations of the CSS values. Inline styles are less performant and will be avoided for styles that are unchanging.

5.10.6 State Management

Managing the state is often done through higher-order components passing properties to children and children emitting events back to their parents. Despite that, the nature of SPAs means each request does not need to recreate all information, such as authentication. Since our homepage offers events at a glance, if the events page had to utilize the same information there is a possibility that the same methods and code would be duplicated. Thus, A global state is required, and while data can be defined on the main Vue instance, the Vuex library allows for global state modules to be made without polluting the Vue state.

Vuex can be utilized in a few different ways, however the most idiomatic and compatible method with Nuxt.js is to use Vuex modules. Each module contains its own state, getters, mutators, and actions. The state is no different than what is returned from the data function in a component, getters are no different than computed properties, and actions are the same as methods. However, Vuex operates on the principle of maintaining a separation of duty and as such, actions can reference the state but cannot directly mutate them, nor can any other part of the app. In order to change Vuex state, mutations must be defined for it. Actions are usually for retrieving data asynchronously from the backend, calling mutators to change the state, and then the rest of the application can utilize it. Vuex is not ideal for all parts of the application, and should only be used if the state needs to be referenced in separate pages or layouts.

An example Vuex module for retrieving and storing events is shown in Figure 43, and how to utilize each Vuex module within a component is listed on the table in Figure 42. When used with Nuxt.js, each module is loaded automatically by naming the file how the module should be accessed in the rest of the application in a store folder. Importantly, actions and mutations can only receive up to one argument. The use of multiple arguments can, however, be simulated by using an object as the only argument.

Component	**Accessor**
State	`this.$store.state.<module>.<value>`
Getters	`this.$store.state.<module>.<getter>`
Mutation	`this.$store.commit("<module>/<mutation>", value)`
Action	`this.$store.dispatch("<module>/<action>", value)`

Figure 42: Utilization of a Vuex module within the context of a SFC.

```ts
export interface Event {
  id: number
  start: Date
  end: Date
}

export const state = () => ({
  status: 'unloaded',
  events: [] as Event[],
})

type EventState = ReturnType<typeof state>

export const mutations: MutationTree<EventState> = {
  setStatus(state: EventState, status: string) {
    state.status = status
  },
  setEvents(state: EventState, events: Event[]) {
    state.events = events
  },
}

export const actions: ActionTree<EventState, RootState> = {
  async getEvents({ commit }) {
    commit('setStatus', 'loading')

    const resp = await this.$axios.$get('/events')

    commit('setEvents', resp)
    commit('setStatus', 'success')
  },
}
```

Figure 43: Vuex module example for retrieving and storing events using TypeScript. The commit methods use magic strings, and are not TypeScript friendly.

5.11 Frontend Framework - Nuxt.js

Nuxt.js, a meta-framework for Vue, provides server-side rendering and other performance and development tools for making websites [31]. Developing within Nuxt is not much different from building standard Vue single-file components, but features like file-system routing, route chunking, SSR, automatic preloading, HTTP/2 header pushing, a PWA module, and a separate modern build for newer devices are available to boost the productivity and performance of the website. In essence, it is a framework built with the express functionality of making Vue websites as production-ready as possible.

Nuxt has three modes of compilation that greatly impact how the website will be developed, how performant it will be, and how it is hosted. The first mode, universal, is the one we will be employing. It renders a Vue website as a single-page application but uses server-side rendering on the first page load. This eliminates a lot of the performance and SEO issues that come with a regular SPA. This mode requires that the Nuxt server be running during production to handle the server-side requests for first page loads instead of files being statically hosted using Apache or NGINX. It is not required that this server runs on the same machine as the backend, however. Next is SPA-only mode, which tells Nuxt to compile the website into a single-page application without SSR. SPAs without this addition are slightly less performant, but have more freedom in hosting and can be rigorously cached alongside newer performance advancements such as HTTP/2 prefetching. The last mode is static rendering, which turns each page into its own document and necessary JavaScript. This mode would not work as the website being developed is far from static.

5.11.1 File-System Routing

Describing the routing for pages within Nuxt.js is done through the filesystem. Within the Nuxt.js framework is a pages folder that will scan each file and map their names to the router automatically. Each file in the pages directory is its own single-file component with some special properties available in the Options API. Some of the special properties and methods provided by Nuxt allow for describing page titles and headers, injecting middleware into the page, and adding new lifecycle hooks to retrieve data and merge it with the state before loading the page.

Dynamic routes are created by including an underscore in the name of the file or folder. Folders with an index.vue file within them will route no differently than if a top-level file with the same name as the folder existed. This is useful for defining a page for a list of entities, and then a separate parameterized route within the folder for an individual entity. Parameters can be validated using a validation method within the page, for instance, if the id parameter is not a number you can redirect to the error page instead of erroneously asking the backend to make an invalid request. The example page routes can be seen in Figure 44.

Example Page Structure	**Resultant Route**
`/pages/index.vue`	example.com

`/pages/about.vue`	example.com/about
`/pages/users/index.vue`	example.com/users
`/pages/users/_uid.vue`	example.com/users/1

Figure 44: Table mapping the pages folder with their router counterparts.

5.11.2 Layouts

Layouts are a means of defining reusable structures pages can inherit. Areas such as the header often do not change across pages, and the less unnecessary re-rendering that is done on each page, the more performant the website can become. They are no different than single-file components, however each layout can be reused by specifying the layout property on the Object API. The admin panel will have a different layout from the primary user layout with the sticky bottom navigation, and the welcome homepage for users who are not logged in will also be its own layout.

5.11.3 Data Fetching

Pages can be developed to either wait for data before rendering, or render immediately and pull data later if it needs to. Large amounts of data or slower requests can be done after rendering to better illustrate to the user that data is loading. Nuxt.js provides an Object API method for retrieving data in pages or on components primarily through the `fetch()` method.

On the first page load, fetch() will be called on the server side, allowing for the website to retrieve data and hydrate it before sending it to the user. This will allow for the site to appear to have been loaded immediately once the underlying requests are finished. However, once the website is loaded, SPA mode will take over and subsequent requests will be loaded from the client's device. The website will have a loading indicator and will wait for the request to finish before transitioning pages, similar to how YouTube does it.

5.12 Calendars & Recurrence

The architecture of a calendar system depends on the necessity of recurring events and its complexity on how granular the recurring events have to be. Projects and courses offered by OMC may recur over the span of a few months, but infinite recurrences are not required. Two primary approaches exist when approaching recurring events: storing

recurring events in a table of event rows for each occurrence, or storing the pattern to be translated into event structures programmatically upon retrieval. Both processes start with a need for some standard for describing recurring events, achieved through a recurrence rule.

5.12.1 Recurrence Rules

Instead of recreating the wheel on describing recurring events, the latest iCalendar standard RFC 5545 describes a structure for recurrence rules called the RRULE, recurrence rule, or recurrence pattern [32]. Many calendar implementations such as Google Calendar, iCloud, and Outlook either use this standard or support it which serves as a model and a means of easily communicating with external APIs. Figure 45 shows the supported parameters of the recurrence rule.

Parameter	Description
`DTSTART`	The start date of the recurrence. This can inclusively be the first occurrence.
`FREQ`	One of the following constants describing the frequency of recurrence: YEARLY, MONTHLY, WEEKLY, DAILY, HOURLY, MINUTELY, SECONDLY.
`INTERVAL`	The interval between frequency iterations. A `FREQ:YEARLY` with `INTERVAL:2` would mean every 2 years.
`COUNT`	How many occurrences to generate.
`UNTIL`	A date that specifies the limit to the recurrences, can inclusively be the last occurrence.
`WKST`	The starting day of the week. Must be either MO, TU, WE, TH, FR, SA, SU or its integer representation (1 through 7).
`BYSETPOS`	Integer or array of integers, positive or negative. Each integer represents an occurrence index, -1 means the last, 1 is the first. For example, setting a monthly frequency, only the weekdays, and a BYSETPOS of -1 would mean the last workday of every month.
`BYMONTH`	Integer or array of integers representing the month(s) from 1 to 12.
`BYMONTHDAY`	Integer or array of integers, positive or negative, representing the day(s) of the month from 1 to 31.

BYYEARDAY	Integer or array of integers, positive or negative, representing the day(s) of the year from 1 to 365. 1 represents January 1st, -1 is December 31st.
BYWEEKNO	Integer or array of integers, positive or negative, representing the week of the year from 1 to 53.
BYDAY	Integer, array of integers, constant, or array of constants MO, TU, WE, TH, FR, SA, SU.
BYHOUR	Integer or array of integers representing the hour(s) of the day from 0 to 23.
BYMINUTE	Integer or array of integers representing the minute(s) of the hour from 0 to 59.
BYSECOND	Integer or array of integers representing the second(s) of the minute from 0 to 60 (rolls over).

Figure 45: Recurrence rule parameters.

Defining recurrence in Google Calendar is done through a dialog window shown in Figure 46. The frequency of repetition, how often to repeat, and when to stop the repetition can be done with a combination of dropdown menus and a date field. They allow repetition on different days of the week using checkboxes with custom styling. Each parameter can be tied to their respective fields and sent as part of the event creation or updating events as the individual parameters. However, the translation of recurrence rule parameters into a valid recurrence string can be done with Node libraries built for this purpose.

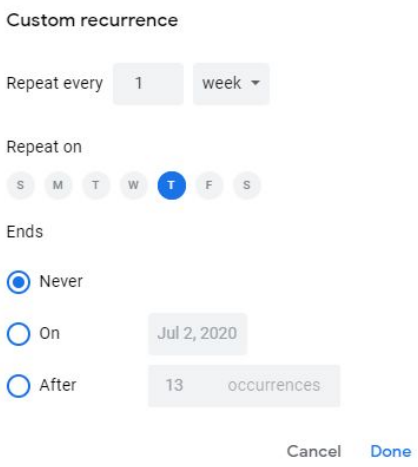


Figure 46: Google Calendar's interface for defining recurring event patterns.

5.12.2 Serializing and Parsing Recurrence Rules

Each parameter of a recurrence rule can be stored individually, however, recurrence rules are meant to be serialized and stored as a string as per the standard. There are exceptions when using certain combinations of parameters and the parameters have a specific order and syntax that can be violated. For instance, the `BYSETPOS` parameter must be last in the rule as the previous parameters are needed to determine the total number of events before referencing the index of the events is possible. The RRule Node.js package can serialize an object of recurrence properties into different structures that make it easy to interact with programmatically. Figure 47 shows a simple example of recurrence serialized, turned into event dates, and also converted back from the serialized string.

```
import { RRule } from 'rrule';

// Every other month on Tuesday and Thursday,
// starting April 1st, for 4 events total.
const rule = new RRule({
  freq: RRule.MONTHLY,
  interval: 2,
  byweekday: [RRule.TU, RRule.TH],
  dtstart: new Date(Date.UTC(2020, 4, 1, 0, 0)),
  count: 4
})

rule.all()
// [
//   2020-05-05T00:00:00.000Z,
//   2020-05-07T00:00:00.000Z,
//   2020-05-12T00:00:00.000Z,
//   2020-05-14T00:00:00.000Z
// ]

rule.toString()
// DTSTART:20200501T000000Z\nRRULE:FREQ=MONTHLY;INTERVAL=2;BYDAY=TU,TH;COUNT=4

rule.toText()
// every 2 months on Tuesday, Thursday for 4 times.

const parsedRule = RRule.fromString("DTSTART:20200501T000000Z\n
RRULE:FREQ=MONTHLY;INTERVAL=2;BYDAY=TU,TH;COUNT=4")

parsedRule.toText()
// every 2 months on Tuesday, Thursday for 4 times.
```

Figure 47: RRule package simple event serialization and parsing.

5.12.3 Generating Recurring Events

The iCalendar format is an abstract structure that does not require relations or other associated data. It has features necessary for defining metadata about events, how events recur, and handling exceptions to events, but nothing further. There are two trains of thought when needing to structure recurring events: store all of the events, or store only the recurrence rule and retrieve the necessary event instances programmatically.

5.12.3.1 Storing all Events

Inelegant yet effective, storing all of the occurrences in an event table results is a straightforward way of dealing with recurring events, but may have performance pitfalls. Creating, updating, or deleting a recurring event that spans a considerable amount of time could tax the server into performing operations on hundreds of entities in rapid succession, potentially leading to scaling problems if these are common. Conversely, databases are meant to efficiently query large numbers of entities, and some developers handle infinite recurrence by storing entities decades or hundreds of years in the future without significant downsides.

5.12.3.2 Dynamically Generating Events

Instead of storing every event, it is possible to store and parse the recurrence rule. Using RRule to transform a recurrence string into multiple event dates, a user can query for a date range and the server can retrieve the recurrence rule from the database and hydrate it into multiple event dates before sending it to the user. Further optimizations can be made by caching the hydrated entities either by storing them in memory or as a materialized view within the database. This approach has the added benefit of not requiring an arbitrary number of events to be stored in the database if the event repeats infinitely.

Conflictingly, storing only the recurrence rule in the database is not sufficient when data associations need to be made on each event instance. Users registering to individual events and paying fees for them pose complications. Registrations could still be tracked, say by a unique date primary key, but this holds no referential integrity. If a recurring event were updated, all related data would need to be updated manually and atomically, adding a lot of cognitive and development overhead to the process.

When infinitely recurring events are required, a compromised approach is ideal: store the recurrence rule and create individual events up to a certain arbitrary date in the future.

Events beyond that point in time can either be created when observed as a byproduct of the retrieving request or generated and cached. The latter approach is ideal if possible as `GET` requests ideally should not have side-effects, and for an event registration system, users should not be registering many months in advance in all likelihood. This approach introduces more moving parts as eventually some process needs to occur that stores event entities. However, OMC does not have a need for infinitely recurring events and thus the simpler approach of storing all recurring events individually can be done to prevent over-designing the system.

5.12.3.3 Event Updating and Exceptions

Considerable investment is required when designing the procedures necessary for updating events. There are different approaches to updating recurring events, however as OMC already utilizes Google Calendar we will be ensuring the same side-effects occur. When updating or deleting a recurring event in Google Calendar a prompt asks if only that event instance, that event instance and future event instance, or the entire history of that recurring event should be modified. Each modification scenario can introduce behaviors that affect past and future modifications.

5.12.3.4 Deleting a Single Instance

When a single instance of a recurring event needs to be removed this is called an exception. RFC5545 has an `EXDATE` parameter that is used for specifying a specific date or dates that should be excluded from the recurrence rule. Once the recurrence rule is amended to include an `EXDATE`, the event instance in the database can be removed, cascading to remove any associating data.

Figure 48 depicts an example RRuleSet class provided by the RRule library adding exception dates. A boolean parameter to the RRule and RRuleSet class allows for the default caching mechanism to be disabled. When enabled, repeat queries, such as `.all()` or `.before()`, `.between()`, and `.count()` will have precalculated dates in memory to reference. It was observed that once one of the querying methods is called, modifications such as adding an exclusion date, may not properly invalidate the cache. As such, caching should be disabled and tests must be written to ensure exclusions are properly handled at all times.

```
const rruleSet = new RRuleSet(true)

// Weekly starting April 1st until May 1st.
rruleSet.rrule(new RRule({
  freq: RRule.WEEKLY,
  interval: 1,
  dtstart: new Date(Date.UTC(2020, 4, 1, 0, 0)),
  until: new Date(Date.UTC(2020, 5, 1, 0, 0))
}))

rruleSet.all()
// [ 2020-05-01T00:00:00.000Z, 2020-05-08T00:00:00.000Z,
//   2020-05-15T00:00:00.000Z, 2020-05-22T00:00:00.000Z,
//   2020-05-29T00:00:00.000Z ]

// Exclude April 8th, and April 15th.
rruleSet.exdate(new Date(Date.UTC(2020, 4, 8)))
rruleSet.exdate(new Date(Date.UTC(2020, 4, 15)))

rruleSet.all()
// [ 2020-05-01T00:00:00.000Z, 2020-05-22T00:00:00.000Z,
//   2020-05-29T00:00:00.000Z ]
```

Figure 48: Using RRuleSet to exclude dates.

5.12.3.5 Updating a Single Instance

Event instances that are part of the same recurrence pattern can be designed to reference a table storing only the recurrence pattern for the event. However, if the date or time of a single recurring event is updated, the pattern is no longer matching the associated events. One solution to this is to separate the event from the rule and treat it as an exclusion, however this would mean any future updates to the rule will no longer apply to this event. Google Calendar does not exclude the date and instead stores the unmodified event date as an attribute that can be contextually attributed to a modification.

5.12.3.6 Updating Future Events

Modifications to a specific event occurrence that should propagate to all future events depends on how the events are to be modified.

- When the date and time of the event is not modified, update queries can be performed on the event occurrences by a starting date and simply replacing the

information. For example, changing the location of future events. This replaces all modifications to existing event entities, but does not recreate deleted event occurrences.

- If the date or time of the events are modified a new recurrence rule is created and used for all of the future events. As the new recurrence rule may have more or fewer events, this is a destructive action that removes these future events and recreates them. Any modifications to future events are lost, and any deleted future events are returned. A reference to the original recurrence rule remains if all events need to be updated.

5.12.3.7 Updating All Events

When all occurrences of an event need to be updated, all modifications and exceptions are erased. Importantly, if a recurring event was by different recurrence rules because the user changed the dates of future recurrent events, all of the recurrence rules are replaced and homogenized. While changing the description of an event's occurrences is fine, undefined behaviors may be introduced if an event date is retroactively changed. Data that references events, e.g. registrations, attendance, or paypal transactions, may lose context if the primary event date is modified.

In order to change all occurrences of an event, each event can maintain a foreign key to the parent recurrence rule even if the current recurrence rule is not utilized. Updates of this manner can query the parent recurrence rule's identifier, make necessary updates, and homogenize the current recurrence rule. If a new date or time is necessary, a new recurrence rule is created, otherwise the recurrence rule of the edited event overrides the others; any leftover recurrence rules are deleted.

5.12.3.8 Modification Erasure

Checks are necessary to prevent unnecessary erasure of data. Event updates will need to scan if metadata was changed, e.g. title, location, description, or if the recurrence rule was changed, e.g. date, time, or count.

- If nothing was changed, no update should be performed as individually updated event occurrences may lose updated metadata.
- If only metadata was changed, all prior modifications to metadata in the path of the update will be lost.

- If the recurrence rule was changed, events in the path of the update will be recreated, overwriting metadata changes and exceptions. It is not possible to reliably map old event occurrences to the new recurrence rule.

If events do not have any relational data, such as registrations, attendance data, or payment transactions, perhaps for being too far into the future, recreating the event is benign. However, recreating an event with relational data is a worst-case scenario. Safeguards and tests will be necessary to ensure that casual recreation of an event is not possible.

5.13 PayPal

PayPal has developed libraries for all of the popular languages and provides an extensive amount of documentation on how to incorporate the checkout feature on a website. On the OMC business PayPal account, the developer dashboard will need to be utilized to create an authorized application to receive payments not directly on the PayPal website. This created application generates an OAuth 2.0 client id and secret used to authenticate the transactions that will be made by the website. During development, PayPal allows a user to create simulated business and customer accounts in their sandbox for testing purposes. These sandboxed accounts implement all of the features of real PayPal accounts and use identical interfaces, only with fake money. Extensive testing of the payment flow will be conducted to ensure that the flow is resilient to errors, and anticipating this, PayPal provides a way of mocking credit cards and generating errors for all steps of the process. This will allow the website to be tested in earnest using sandbox credentials.

There are many libraries which incorporate PayPal's API in order to handle payments, often to simplify eCommerce, however we will only be implementing a single checkout flow for event fees and will not require advanced behaviors. However, if the application were expanded, for instance to list recent payment issues on the admin panel, or to provide a more comprehensive expenses dashboard, PayPal offers separate tools for this purpose. For checkouts, PayPal offers the Smart Payment Buttons system by providing a tailored script for each application and seller. The buttons, depicted in Figure 49, are loaded by a script provided in the documentation that requires the application's client id in the URL of the imported JavaScript file in order to authorize the website as the seller. Once loaded, the constructor is called which requires two methods to be provided — one

for creating the order and one for handling the transaction, PayPal then takes over and ushers the process themselves.

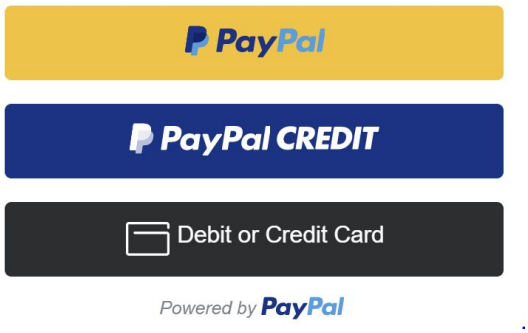


Figure 49: Smart Payment Buttons as shown in the PayPal API documentation.

5.13.1 Creating Orders

The first part of the PayPal checkout process is defining the item being purchased. This is referred to as setting up the transaction in the PayPal documentation, and is done using the `createOrders()` function function of the library. The minimum required configuration needed in this function is a call to the passed action method to create an order, then the number of purchase units can be provided. By default, the library operates in USD, and will not need to be configured.

When we retrieve the status of event registrations, a join will need to be made to check for the presence of an already completed transaction for this event, as well as ensuring that the field that pertains to the cost of the event is selected. While the client-side code could be tampered with to trick the website into believing that the event lacks a fee, the server will perform its own validation to prevent this kind of abuse.

5.13.2 Handling Transactions

Once an order is created, the onApprove() method must be defined. This method returns successful transaction information. Instead of allowing the client to handle this method, a promise to a server method can be provided to handle the successful response on the API. It is there that the transaction would be stored appropriately in the database and a payment confirmation success response returned to the user.

If the transaction fails due to the card being declined or other information was invalid, an `INSTRUMENT_DECLINED` error will be sent to the transaction method. PayPal provides a

method for restarting the transaction and describing to the user the nature of the problem using the `actions.restart()` method. If there are external issues with the payment, such as a chargeback or payment dispute, OMC will need to handle these manually through the PayPal website.

6. System Design

The system architecture will consist of dual broadcasting server instances and a PostgreSQL database on a linux system. NGINX, a reverse proxy server, will map requests from the main domain and API subdomain to the appropriate frontend and backend ports as well as provide some caching and SSL abstraction.

- The website frontend will be a server-side rendered (SSR) Vue.js single-page application (SPA) using the Nuxt.js framework that allows parents to register, sign up for events, pay for event fees using PayPal, configure email notification preferences, and register other family members to the account. An admin panel will also be created on the frontend that admins will use to manage users, create and modify events, modify volunteer hours, and send emails. The frontend will be designed mobile-first, but will responsively support desktop resolutions.
- The REST API backend server will consist of a NestJS framework that wraps the Express.js library. The backend will persist and query data in a PostgreSQL database located on the same system.
- The account system will be similar to the Netflix profile feature, allowing multiple tenants within a single account to have separate experiences without requiring each to have unique registrations.
- Authentication will use a two-tier guard system, first ensuring that the user has logged in with proper credentials, and secondly restricting website access until a specific user is selected through the user picker.
- Authorization will use role-based access control (RBAC). Admins, Volunteers, Educators, and Users are the primary roles, separated by mechanical processes such as the volunteer form being approved by an admin. Age, grade-level, and gender are the primary attributes that describe access to events which will be checked on runtime for the proper permissions.

6.1 Environment Variables

The application will utilize environment variables to configure the frontend and backend. Extensive documentation will be included on how each environment variable operates as well as reference any overridable framework or database specific variables we do not directly utilize. Similarly, both the frontend and backend will validate the validity of the given environment variables to lessen the risk of a misconfigured system. There will be two env configuration files to prevent confidential variables being loaded into the frontend. Allowable configurations typically fall under the categories of credentials and routing information.

1. Credentials: The backend will need to understand how to connect to the database, payment gateway, and emailing services as well as what secret to use for hashing authentication tokens.
2. Routing Information: These variables pertain to which ports the frontend and backend should broadcast on, the domain of the frontend and backend, and if the backend and frontend are on the same system (to allow for localhost communication during SSR requests).

An object schema will be used to define how each property should be configured, if it is required, and the type of the value. If validation fails, the respective server where the validation has failed will not start and explain why in the console.

6.2 PostgreSQL Database

The ERD representing the database schema and relations can be found in Figure 50. MikroORM will join entities using virtual properties, or properties not present within the database, denoted by a “V” in the figure. These virtual properties are to be defined within the entity files of the backend for MikroORM to join relations using JavaScript object structures.

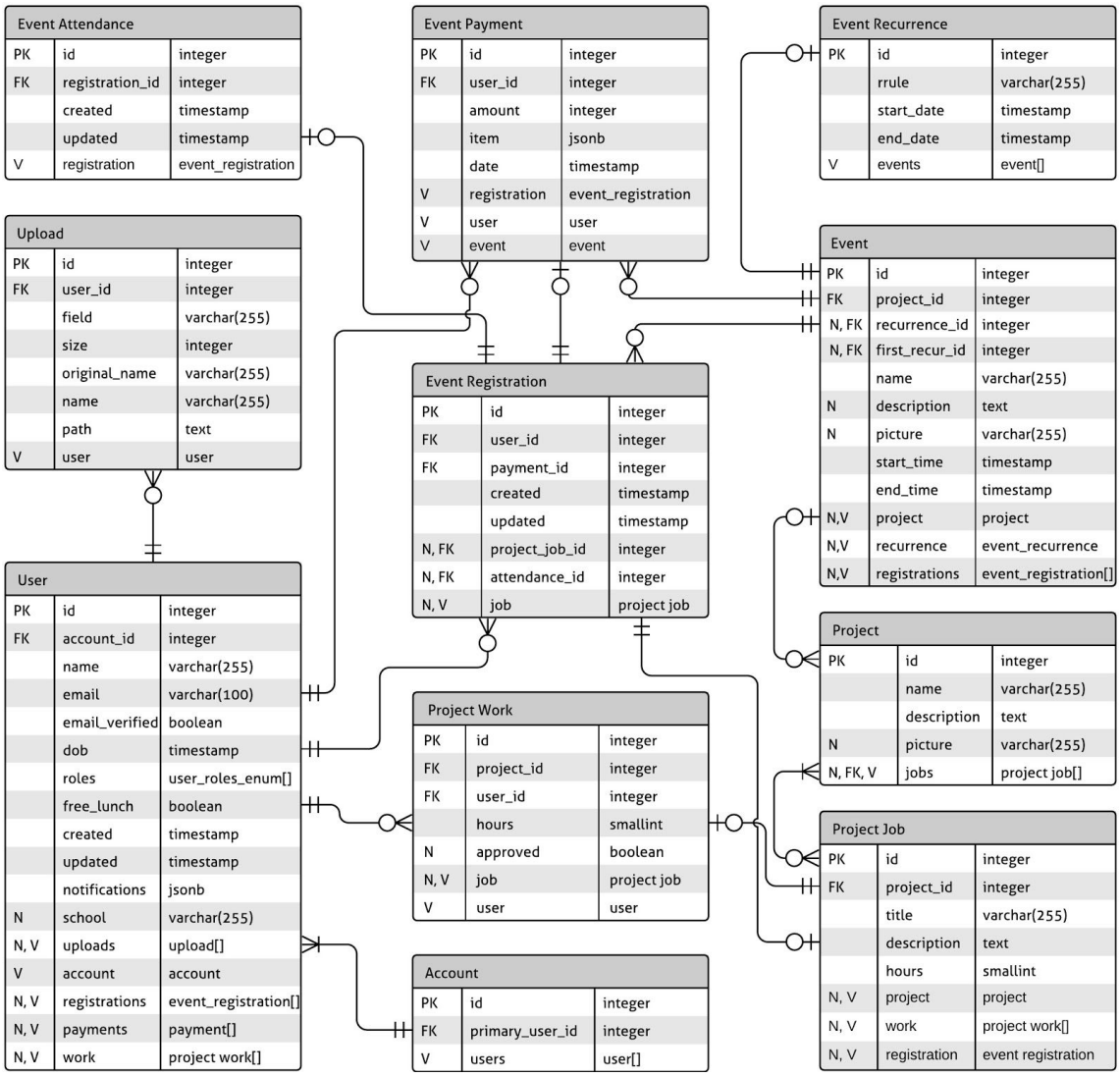


Figure 50: Database ERD diagram with virtual properties shown.

How each entity relates and is created is covered in the backend section. Below is a short synopsis on what each entity and how notable properties within them are created and what their purpose is.

- Accounts have a singular primary user and many user relations consisting of at least the primary user, but are intended to have the other child user accounts.
- Users can have multiple uploads for approving reduced lunch forms. Creating this form sends an email to the appropriate administrator(s), wherein the administrators may review the form and change the free_lunch boolean on the user account. The methods for approving or removing an upload can be

configured to optionally delete the file. This entity is verbose and generic for future-proofing.

- The notifications property on the user entity will be the queryable jsonb datatype. This grants more freedom to define more granular notification preferences on users. It will allow multiple notifications to be defined per email type, and removal of the field hides it from the email scheduler's query, disabling notification emails. This object will also define if the user has a hard email bounce and count the number of soft bounces received.
- Events can exist on their own or be related to an event recurrence defining a string of recurring events. Events can also be part of projects, and projects potentially have multiple jobs. These are all joined collectively in most operations as registration will show jobs to volunteers, and the project information near the event information.
- Registration to an event can either occur on its own, or by first requiring that an event payment exists first. Educators are able to see events they are not themselves able to register for in order to promote the events to their students.
- Event attendance records are created and manipulated only by admins. If the user was not registered, when admins insert the attendance record it will cascade-insert a new event registration. This form of event registration also bypasses the email verification step.
- Volunteers see form fields during event registration allowing them to pick any predefined jobs that are inherited from the project. A project work entity will be created by checking if the registration event has selected jobs during the attendance creation process. Administrators are able to edit the work done in the admin panel if the volunteer left early or was not able to complete their job. It is possible the attendance process is expanded to include checkout times, if at least only for volunteers.
- The project page will detect if the user is also a volunteer and give them the ability to submit a request for volunteer hours to be approved by the administrators as not all work is strictly related to any individual project or event. It partially fills out a project work entity but defines the usually null approved field as false, excluding it from most project calculations until the admins manage it.

6.3 Frontend

The frontend will be designed to feel approachable with as close to a native mobile experience as possible while focusing on accessibility and usability. To improve upon the lengthy registration process of the previous application, registration will be short, allowing for events to be viewed and other users added to the account without email confirmation. Signing up for events on the website will require an email confirmation, however the admin-only attendance page will allow for registration overrides in-person if they have not completed the process.

The website will be a Vue.js SPA with two primary layouts, the one users see and the one admins see when managing the website, referred to as the admin panel in this document. A global state will be managed when the website is loaded to persist data through page changes and to communicate with the backend. While server-side rendered, the frontend will not be coupled with the backend thus allowing it to be hosted separately, modified irrespective of the backend, and promoting maintainability. Thanks to Nuxt.js, every first page load will be hydrated on the server with subsequent browser requests being done on the client's device to improve time-to-paint, responsiveness, and limiting the need for loading placeholders.

6.3.1 Libraries

The frontend will be built with the following Node.js libraries and tested in user trials and in conjunction with end-to-end testing done through continuous integration on GitHub.

- **Nuxt.js**: A Vue.js framework that provides the component building file structure, page routing, server-side rendering, and compilation tools.
- **Vuetify**: Component library based on Google's Material Design specifications.
- **Vee-Validate**: A client-side form validation library for Vue.
- **AVA**: Utility for performing end-to-end testing on the frontend.
- **Vuex**: State management library that provides a centralized data store.
- **Axios**: A HTTP client provided through a Nuxt module that stores base URLs for client and server side requests with auto-retry and request cancellation features.
- **TypeScript**: Adds typings to JavaScript helpful for describing data structures, catching development errors, and warning developers when operations are done on possibly missing data among other things.

- **Eslint & Prettier**: Eslint enforces a coding standard by linting code and Prettier allows for automatic code formatting to meet the standard.

6.3.2 Wireframes

The authentication gateway and user areas make up the primary layout, modeled roughly using wireframes in Figure 51. When navigating to the domain of the website a captive welcome screen has the user either register or sign in to the website before continuing. Navigation needs to be as simple as possible, and users will be most accustomed to a sticky bottom navigation present in native mobile applications. The top of the page will have a non-sticky app bar that can be used to perform quick actions on the user account, such as switch users or logout. These functions will be redundantly available on the account page for users more comfortable using the bottom navigation.

The admin panel wireframes are located in Figure 52. The admin pages will utilize a middleware that immediately redirects unauthenticated users. Once the pages are loaded, each request will be amply protected on the server and will return a `403 Forbidden` error that the frontend will use to redirect to either the homepage or the error page. Even if the user is successful in entering the admin areas and disabling this redirection, only the skeleton of the page will load with no information. The admin panel will utilize a hamburger-menu style navigation for each page with management typically being done in large filterable tables that take up the full width of the device they're viewed on. Each table will have a different level of granularity to search and filter by various attributes. Smaller edits can be done using dialog windows, whereas full-entity management will need to be done on its own page. Component design will take this into account to ensure the maximum amount of reusability.

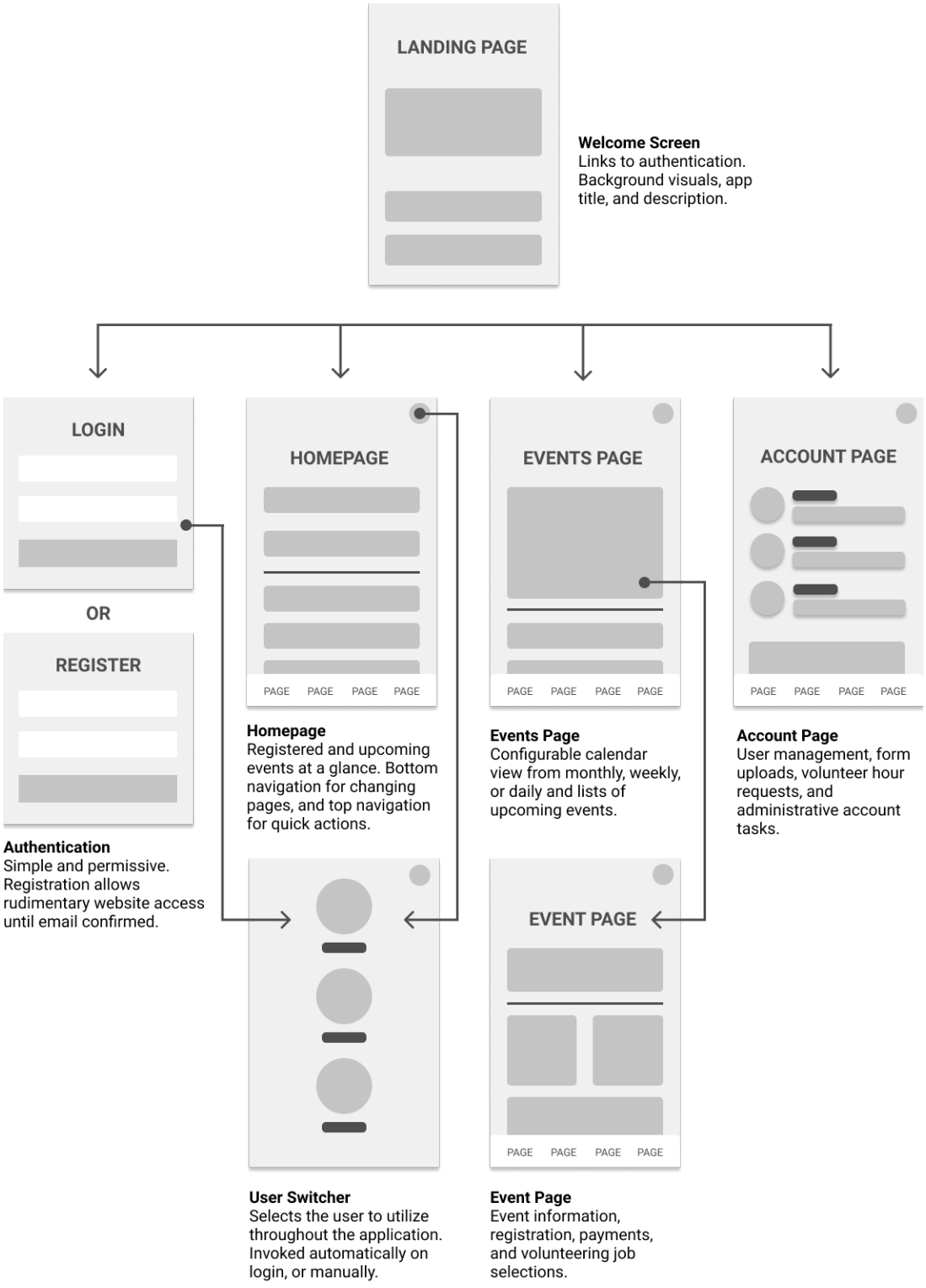


Figure 51: Website root and user page wireframes.

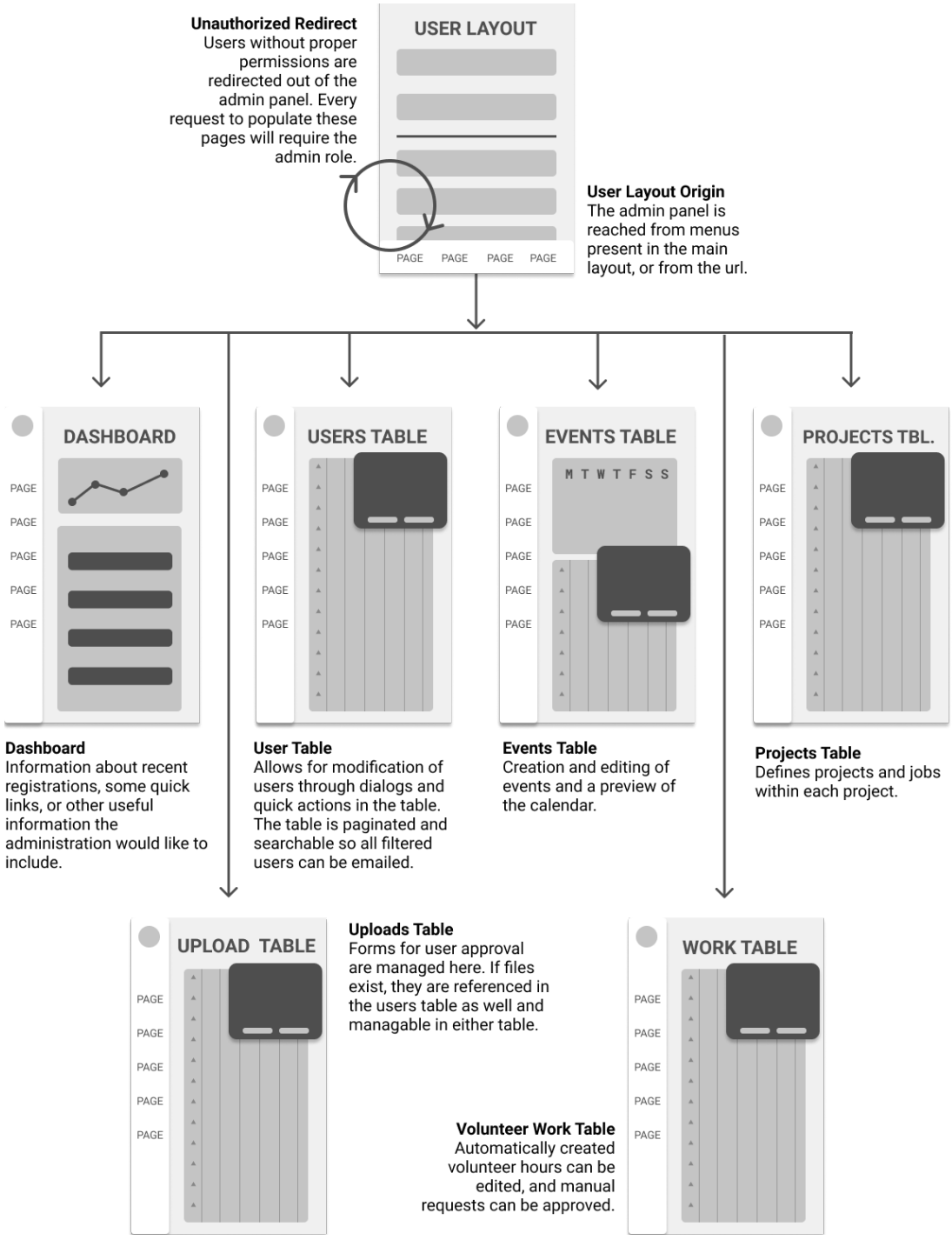


Figure 52: Wireframe mockups of the pages of the admin panel.

6.3.3 Authentication

After a successful authentication or login flow, a token is returned back to the user that will be stored in a Vuex module. Subsequently, every future request to the backend will also include this token as a bearer header to provide authentication to the backend. Figure 53 describes the process, starting with the Vuex module on every first page load. Vuex hopes that the token exists, but if it doesn't the token property is never updated and the default falsy value remains.

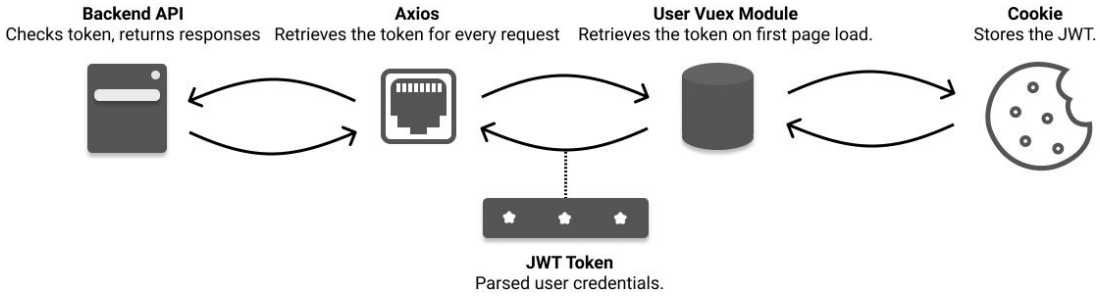


Figure 53: Model of the token handling process for authentication.

On every Axios request to a URL that contains the backend's baseURL, Axios will use an onRequest() interceptor to check the Vuex module for the existence of the token, and if it does, a header of the form `Authorization: Bearer <token>` is attached. During the request, the backend JWT strategy will only look for this bearer token in the authentication flow. Any cookies that are automatically passed to the backend are ignored in the process. While localStorage could be used in lieu of a cookie, and is a common storage mechanism for SPAs, reauthentication from a stored token is almost always going to be done during server-side rendering. This is possible because the first user request will still forward the cookie to the frontend server like how a typical httpOnly session cookie would be utilized.

Logging out a user is done by removing the JWT cookie and purging the user data. Due to reactivity, the website will notice the lack of a user and re-render each page as if they were logged out. A secondary logout mechanism on the account page will allow the user to logout everywhere by invalidating part of the secret used to hash their token on the backend. When this occurs, the next subsequent page refresh will return unauthenticated (typically on the request to get the user profile) on other devices, and a global middleware will catch `401 Unauthenticated` errors of this form and remove the token and user all the same.

6.3.4 Interface Prototypes

Several prototypes were built using mock data for the frontend on Vue, Nuxt, and with the base stylings of the Vuetify component library. These prototypes focus on showcasing core components of each page as opposed to the visual. Much of the website building phases will consist of rounds of visual prototypes and accessibility testing. The reactivity provided by Vue also allows for styles to be replaced easily to make the website more engaging looking for children.

6.3.4.1 Homepage

The first prototype is the homepage shown in Figure 54. This page represents a portal for returning users to view the status of upcoming events, their times at a glance, and providing a seamless way of transitioning to the more utilitarian features of the website. The cards, or the components which are listing each event, will likely be more condensed with a divider feature for making the date and time much more prominent to users. The homepage also showcases the app bar, housing a button for some quick user actions. This area will likely be partially expanded to include the page's title and a way for admins to access the admin panel if they have the appropriate roles.

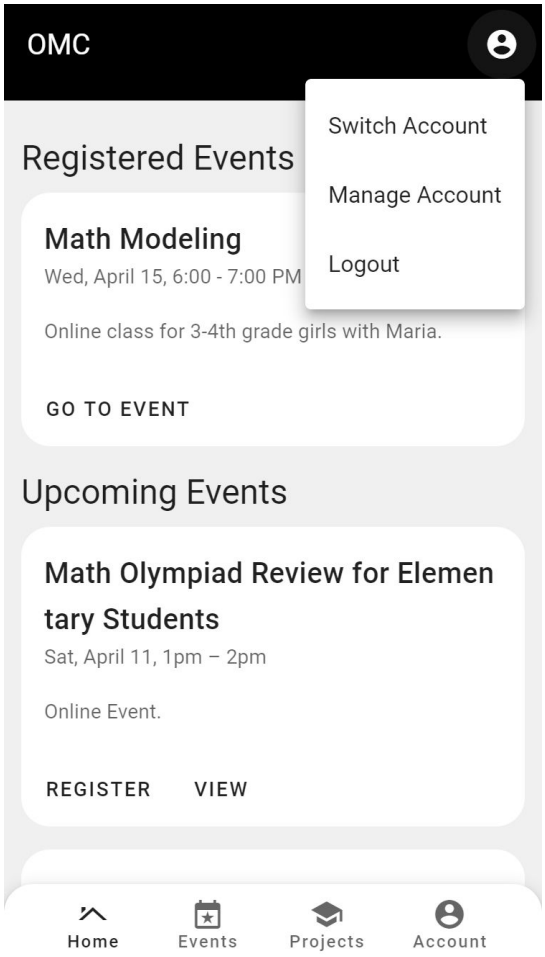


Figure 54: Homepage prototype for listing registered and upcoming events.

6.3.4.2 Events Page

Users will be able to visualize and sign up for events on the events page. The page will have a large calendar and a list of events below it. As the user changes the date range by using the arrows the website will query the backend for all events in that range, the requirements for viewing and registering to the event, and any registrations or payments made for these events. By default, the layout will display as a weekly calendar view but users who have larger devices or are more technologically adept can switch to the monthly view, and users who require a more granular view can switch to the daily view. A cookie will be persisted on the users device that maintains this setting.

The calendar will need to overcome many accessibility, real-estate, and browser limitations of being a website. For example, most calendar apps allow for free-horizontal scrolling that will be explored, but mobile devices often use the sides of the device for back and forward button behaviors that could lead to a frustrating experience. Similarly,

while events can be accessed by tapping on them, this functionality may not be desired for all users. However, allowing the calendar to grow pushes the event list far enough down that users may not know it exists, yet capping the height of the calendar may not prove sufficient if the device is small enough. Thus, it is likely the calendar and events list will be split into multiple pages, however event logic will need to carefully span the homepage, events, and calendar pages, increasing the possibility of confusion that will need to be tested. Figures 55-57 illustrate the four possible views of the calendar component: monthly, weekly, 4-day, and daily views.

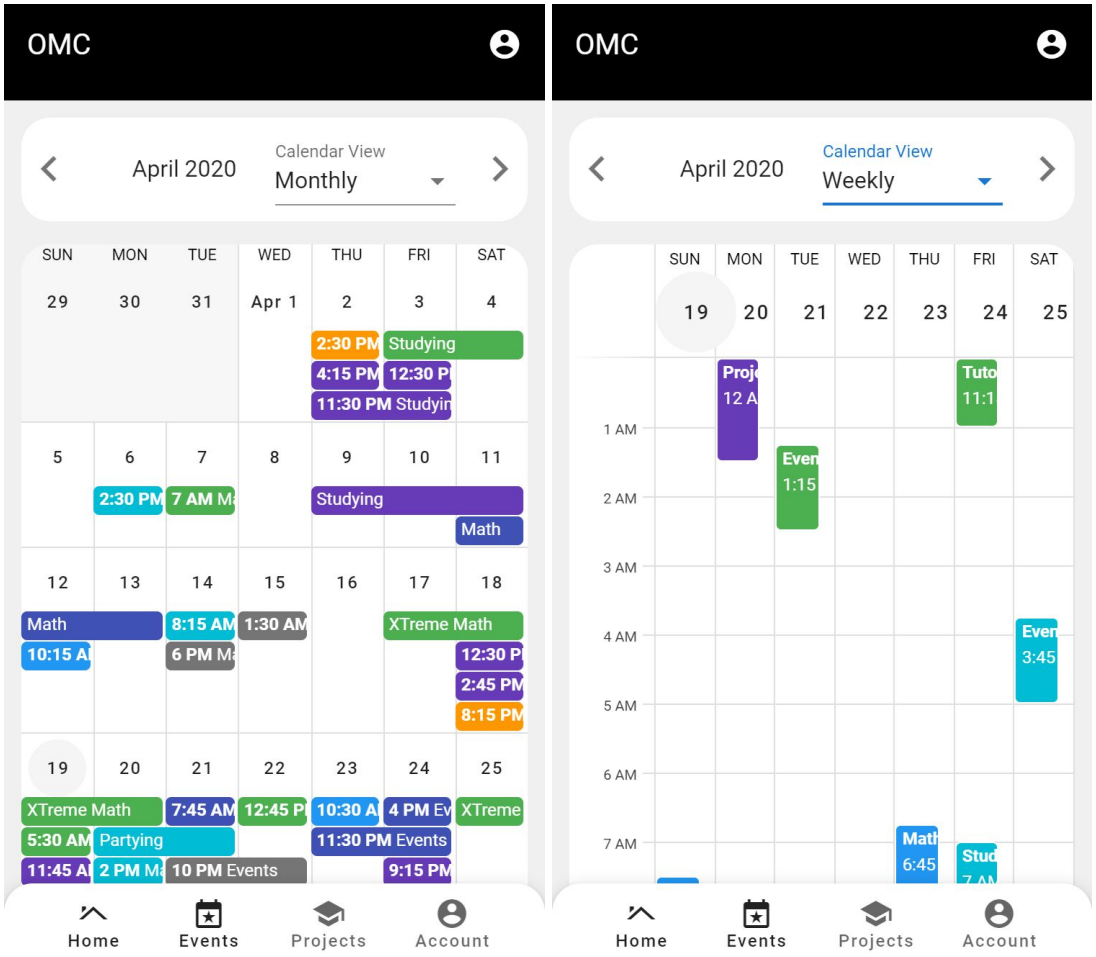
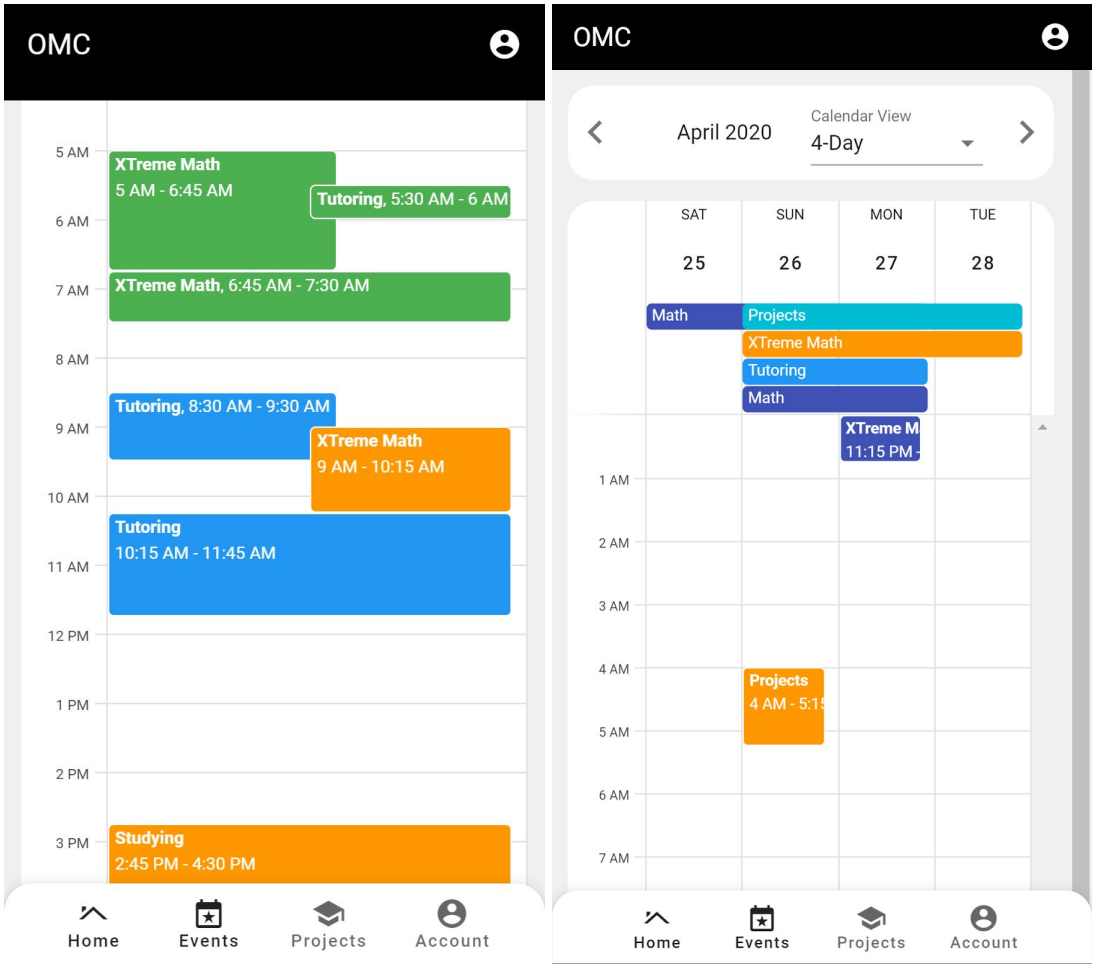


Figure 55 and 56: Monthly and weekly prototype calendar views.



Figures 57 and 58: Calendar prototype for day and 4-day views.

6.3.4.3 Event Registration

Once the primary user has added others to the account and verified their email they are allowed to register for upcoming events and pay any necessary fees. Any reference to an event be it from the homepage, calendar, or list of events will allow a user to view the event page. The amount of content on the event registration page largely depends on the roles of the user, some of which are seen on the prototype in Figure 59.

- Administrators will be able to see the current allotment of registrations to the event and the list of all registered users.
- Volunteers will see a list of jobs for the event they can optionally select during registration.
- Educators will see events that would not otherwise apply to them.

If the event is part of a project, the event information card includes the project picture if there is one, a description of the project, and then a description of the event if it has a separate description. Events will likely also have hosts and location information alongside the description to supplement the timing. Below, users within the account that fit the criteria for the event can be selected during registration, or edited and added at a later point in time, or the registration withdrawn entirely.

The PayPal Smart Buttons will appear on this page as well if payment is required. It will be up to the admin which made the event if payment will be required to complete registration, and if so, the event will grey out a button at the bottom of the page to submit the registration until the payment flow is completed. Otherwise, a radio button will be present denoting that the registering user is affirming they will be paying in cash at the event. Though obscured, PayPal accounts will not be required and the payment flow will expand to allow payment information to be entered directly. However, if the user does not want to re-enter this information, a PayPal account will be required. Successful registrations will change the page to include a status of the user's registration and payment confirmation.

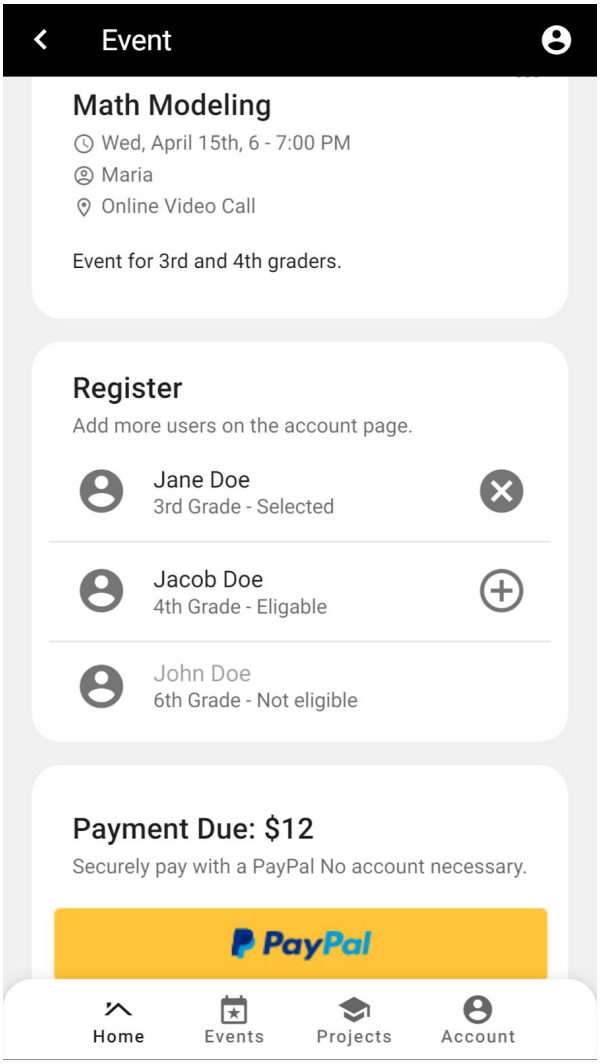


Figure 59: Prototype event registration page.

6.3.4.4 Account Page

Interactions users have on the website are all centric to user management. Even the event systems are overlaid with user records like attendance, registrations, and payments. Management of the multi-account system and the users within them will need to occur at the account level. However, much of the design consideration for users comes with the fact that not all users are first-class citizens within the application. Children added to accounts can be selected, however the account page for them would not show payment information, forms, notification settings, or how to manage other users. Particularly, the only feature child users have access to is the information about how many events they have been to, and if implemented, a progress bar for points earned by attending events. Due to the reusability of the components and the ease in which reactivity can be

incorporated in the website, it will not add a considerable amount of development effort to create completely different experiences for different user roles. To illustrate some of the planned features of the primary user account, a prototype was made that can be seen in Figures 60 and 61.

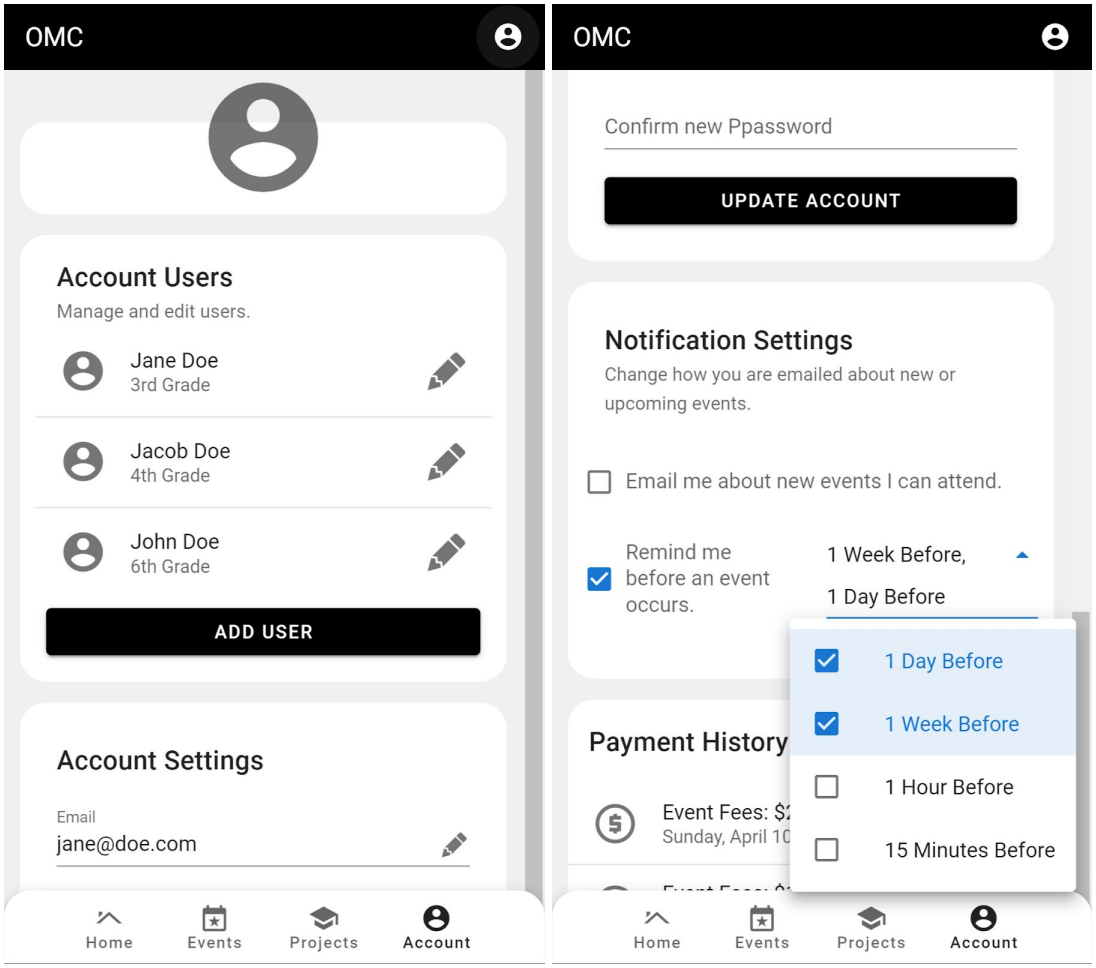
User management from the primary user is going to be a high-traffic account feature. So much so that when an account is first created and logged in, a temporary property is sent with the token to instruct the frontend to include some light instruction to focus on the account management area. A banner that is meant to catch the attention of users will inform them to add children to the website, as otherwise, registrations will not be possible. Other banners will be present on the account page or globally, for instance, an unverified email will create a banner present across all pages to inform the user to complete the process.

Account settings will allow the user to change their email or password, and both of which will require transactional email verifications. However, a descriptive means of allowing users to receive notifications is important to maintaining OMC's reputation with AWS SES, increasing the chances of deliverability and not annoying users. Overall, emails will be opted into with the easiest possible means of opting out or reconfiguring their delivery. Some other emailing examples include confirmation emails of payments as part of the PayPal process, however users may wish to see their payment transaction history. A paginated table will show the most recent few payments, if there are any.

The reduced lunch form, shown in Figure 62, is a specialty client requirement as part of a manual approval process for denoting accounts as having free or reduced lunch. This process is the most likely to be amended or the delivery mechanisms explored more carefully. It is not expected that the user demographic will understand how to upload documents to their phone, posing a potential usability issue. However, the website will operate normally on desktop devices, so some level of instruction may be necessary. For example, it would be possible to email the user a set of instructions, or give the user a PDF containing information about how to typically retrieve the form from schools, how to scan it, and then how to upload it on a desktop computer.

The volunteer handbook will also be signed electronically through this page. The process will only be possible for the appropriate age-ranges, however the delivery is likely either a PDF or other form of document that the clients want the user to read and affirm they have read and understood the document. Some features that can be explored here is

disallowing the submission of this request through the UI if the bottom, or close to the bottom, of the element that contains the text for the volunteer handbook agreement document has not been reached. A binding affirmation through the use of a checkbox is the current implementation idea, however.



Figures 60 and 61: Prototype account page user management and notification settings.

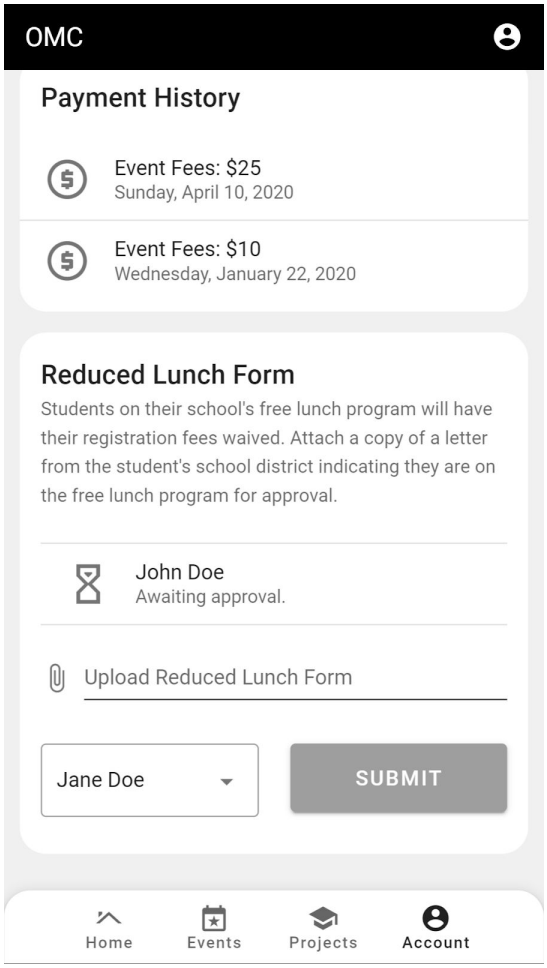


Figure 62: Account page reduced lunch form input and approval.

6.3.4.5 Account Switcher

The utilization of a multi-tenant account structure is a gamble in usability. With the old system, children had no purpose using the application. Allowing children to have accounts is commonly handled by other websites such as Google by requiring an interlocking confirmation process where the first user has affirmed their age to be above 18, then the child user can be referenced through a special offshoot of the registration process by sending an email to this parent user to approve of the creation of the account. Google may also be afforded special freedoms, as current recommendations whenever information from children is potentially collected, such as them entering their own name into a registration form or sending emails, confirmation through driver's licenses, signed documentation, or other means is recommended and are not easily validatable for this website. This interlocking account system would also involve an email or username and password for every user on the account, overloading the parent in such a way that likely led children to create accounts in their parents stead to begin with. Thus, this account

system is predominantly managed by the only admin account user but can share authentication for others such as how Netflix styles their profile system despite being under a singular login.

An implementation mockup for the account switcher is shown in Figure 63. The visual specifics of the selector can be made to expand or contract depending on the number of users on the account. However, this page has two different modes of operation. First, when a login is made and the token received is incomplete, or there are multiple users on the account and the user needs to affirm who to browse the website as, this page will become captive and not let the user do much of anything else besides logout. If the user is invoking the account switcher having an already complete token, the process can be aborted, and thus the sticky footer would reappear and a back button would be present in the top app bar.

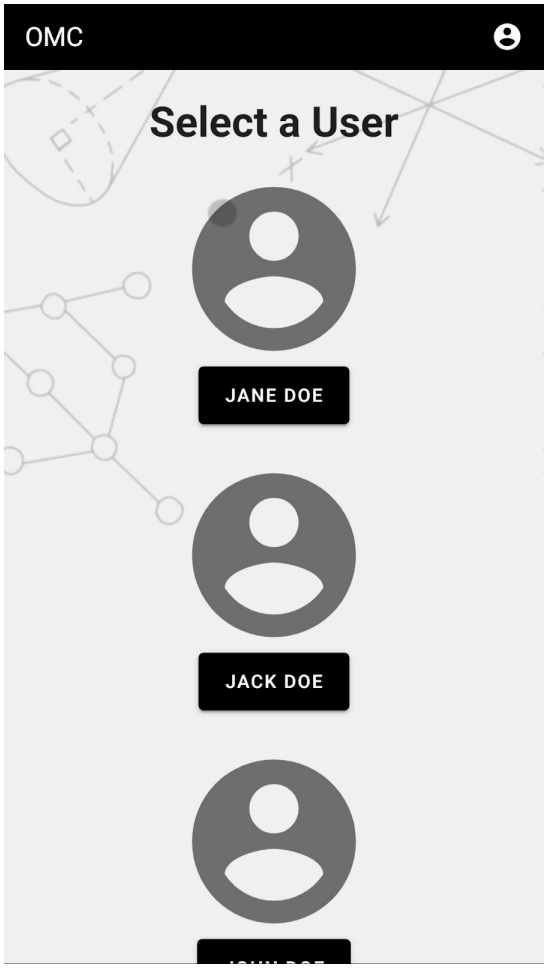


Figure 63: Prototype account switcher keeping the user captive.

6.3.4.6 Admin Panel

The admin panel is an important part of the application as a form of simplified content management system (CMS). The structure of these pages is a collapsible navigation drawer that houses the user actions and the links to the other pages for managing entities within the panel as seen in Figure 64. Aside from the dashboard, a homepage of the admin panel for minor statistics or links, the other pages of the admin panel have the same tabular structure. Management of entities will either occur through a page specific to the entity that is structured to allow changes to all of the entity fields, or through quick edits from dialog boxes.

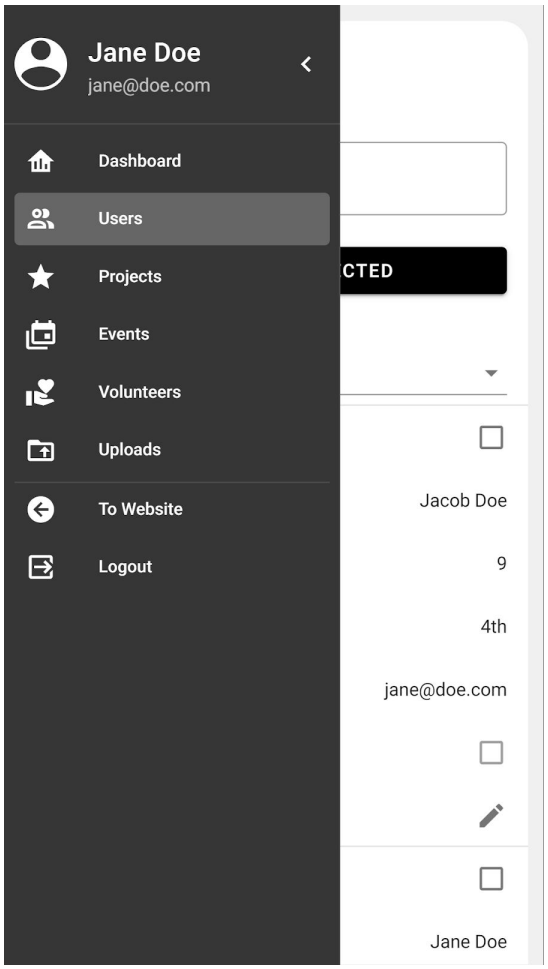


Figure 64: Admin panel popout navigation menu.

Figure 65 shows the prototype users management data table for a few mock users in its expanded mobile interface. For the best experience, management will also be supported on desktop devices in a typical table view, shown in Figure 66. Both versions of the table will support pagination, sorting, selecting individual entities, filtering by properties, and editing individual entities. Entity-specific quick actions, such as previewing the uploaded documents for free lunch forms, approving free lunch forms, and editing volunteer work hours will be possible without needing to open the entire entity. Edits made will prompt a snackbar, or small popup dialog, to appear to inform the admin of successful (or unsuccessful) modifications, as well as every button utilizing loading indicators.

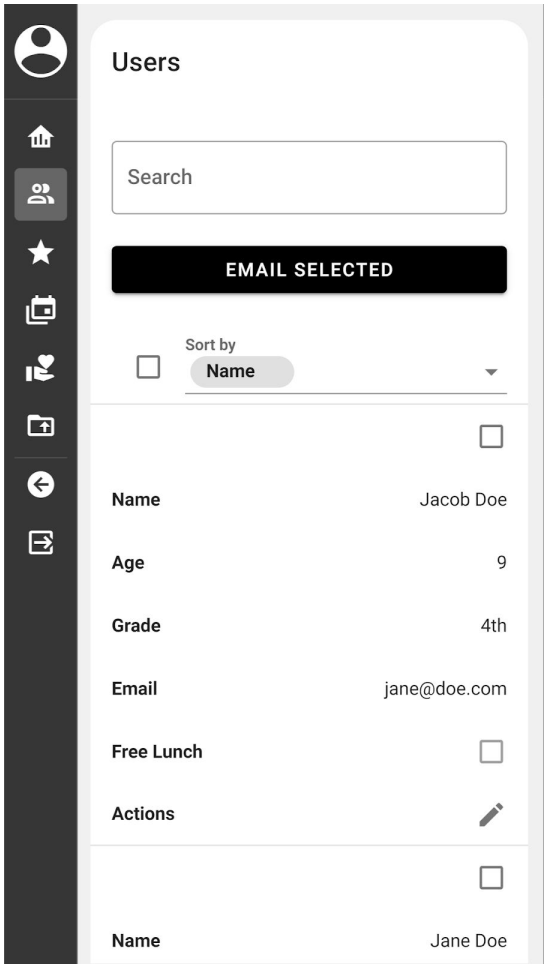


Figure 65: Prototype user management admin page and table.

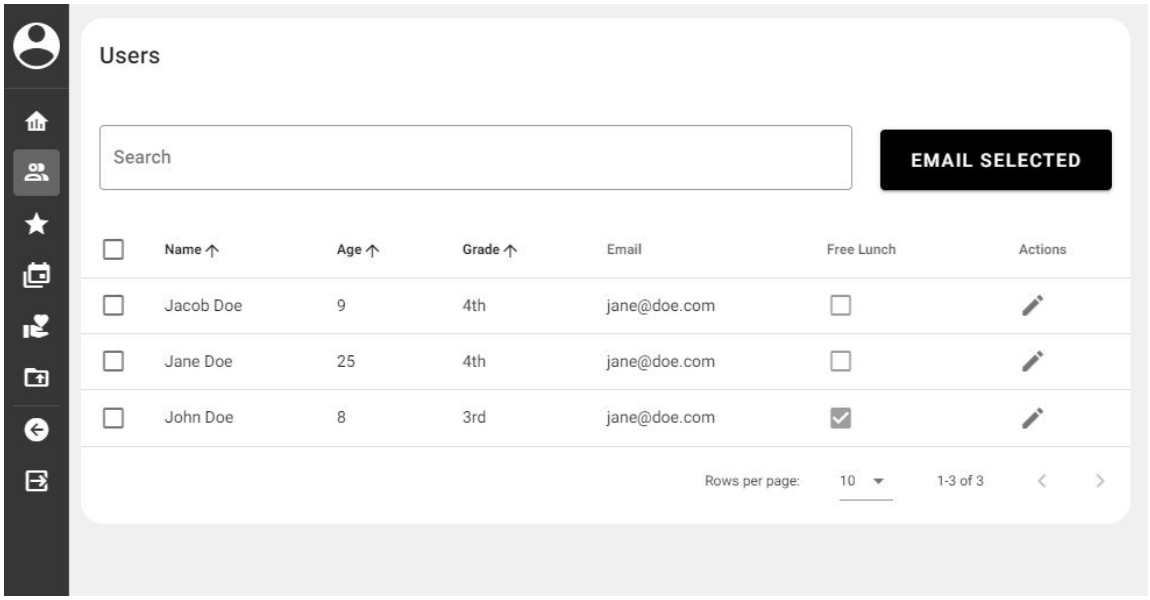


Figure 66: Prototype user management admin page on iPad or larger resolutions.

6.3.4.7 Attendance Page

When users arrive at events, some form of attendance tracking is to take place. The core functionality is likely to be similar in principle to table-based pages such as in Figure 66. The ideology for how this page is utilized has been left to the current discretion of the team, and as such we are interested in testing an admin-only page that operates as a check-in system with the ability to override registrations. The alternative to this would be a guided user check-in system. However, the former would allow for admins to override registrations and take cash payments in person whereas the guided user check-in system would require for users to make an account on the website, add their children, and then register through the application before being able to check-in. This admin check-in would also be able to act as a means of automating if volunteers showed up, and potentially with the ability to mark when they left, if a “check out” feature was desired to count the hours automatically.

Regardless of the style of the attendance page, it will utilize websockets to refresh information dynamically. In terms of the interface, animations will be used to update the interface in such a way that users are aware that it has refreshed, is not jumping and interrupting the use of the page, and is not an obtrusive message. The list could grow downwards by registration time, and sortable by name for those having a hard time finding theirs. If users are not part of the event registration, they should be able to search by their email and register themselves and check-in to the event simultaneously.

6.3.4.8 User & Admin Login

The user login form will be part of the flow out of the captive welcome portal. The interface itself, seen in Figure 67, will not have any atypical features from any other login form. The form itself will be made into a reusable component capable of being used in a dialog window or other standalone page. This is because when the admin page is loaded without any user token being found, to save the admin time the login form will be redirected to. However, the primary user login form will redirect to the user layout homepage, adding more steps for administrators. Thus, a secondary login form under the admin area will be utilized which redirects into the admin panel. Functionally, the login components themselves can be identical, however a slot can be used to override the links below the password field, for instance to not include the sign up link but instead replace it as a means of returning to the homepage.

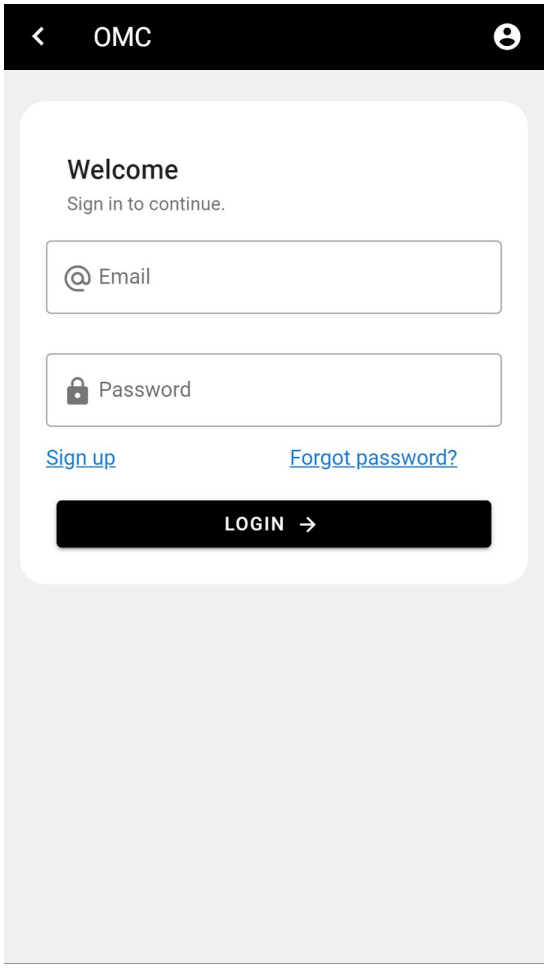


Figure 67: Prototype user login form.

At the end of every authentication flow is a call to a route which will retrieve the user after initially storing their token. Any non-admins that used the admin login process would be found to have the wrong permissions and be ejected from the area using a middleware.

6.3.4.9 User Registration

One of the requirements most important to the clients is the need for as quick of a registration process as possible, a mockup of which is in Figure 68. Their old application had a 7-step registration process. We are going to be trying to condense the number of steps from a user registering to having successfully registered to their first event in 4 steps.

1. Registration Form. The form will ask information that is required to create the primary user and receive consent to send notifications, a requirement of AWS SES and other reputable email providers.
2. Adding Users. Immediately following a successful registration (email is not taken), the user will be asked if they would like to start adding users to the account and warn them that they also need to confirm their email address.
3. Email Confirmation. The order of this step and Adding Users is unimportant; users can be added to accounts without email verification.
4. Event Registration. The user can now either check-in through the attendance process at the event itself and have their registration created that way, or they can register through the website and check-in quicker.

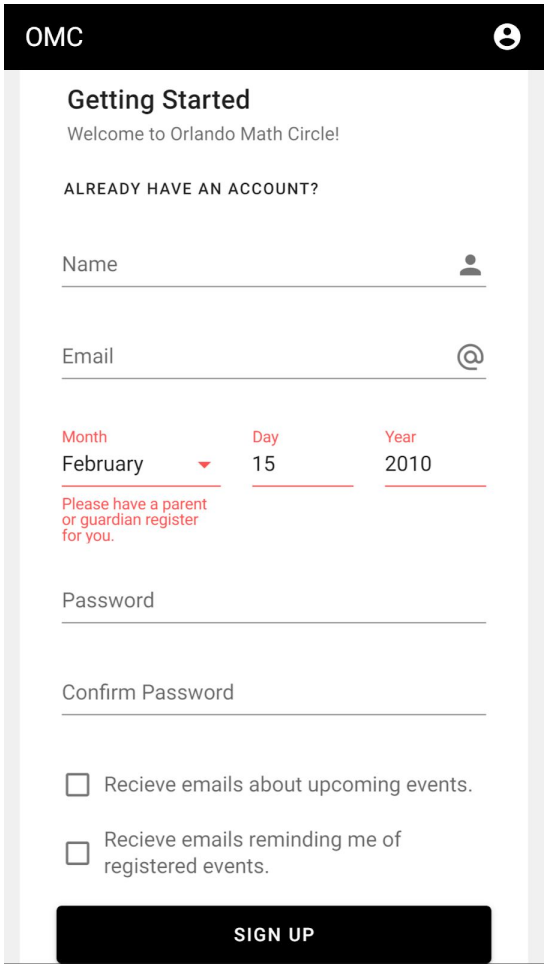


Figure 68: User registration prototype with example error handler.

6.4 Backend

The frontend will utilize a RESTful API to query and transport data from the database. While the website could be monolithic in design by packaging the server and interface components into one program, this approach sacrifices extensibility and maintainability. Client-server separation, statelessness, cacheability, and hierarchical design philosophies of REST promote highly extensible and reusable systems ideal for this website and the students which will learn from it [33].

Each resource in the database is simple in scope, but each is highly connected. The API will be tasked with determining the level of access control for each actor and carrying out approved actions on resources. Not all users may register for events, not all events are visible to every user, some users can bypass payments, some actions may require admin

approval, and admins should have near universal access to all available resources and other actors. The backend will also utilize a task loop that queries the database for events users are registered to and sending emails as appropriate.

6.4.1 Request Lifecycle

Each request on the backend will follow the following lifecycle. Services that utilize queues do not return anything and process their tasks asynchronously outside of the request flow.

1. Global Middleware: Rate limiting and logging.
2. Guards: Enforce authentication or authorization on the controller or a controller route.
3. Pre-Controller Interceptors: Starts processes before routes, optionally finishes them after, e.g. starting a method timing how long a route takes to execute, or intercepting uploaded files before passing along the parsed body.
4. Pipes: Transform or validate inputs. The validation pipe occurs at this step, ensuring the body, parameters, and query parameters match the types in their associated data transfer object (DTO) or the request is stopped.
5. Controllers: Scaffolds the routing, parameters, guards and pipes before sending the payload(s) to the appropriate service. The controller body determines the permissions of the user and picks the appropriate service method.
6. Services: Perform the request logic and query the database through MikroORM. Each service and the database repository that feeds it are encapsulated within a module, performing operations solely on the entity for its repository.
7. Post-Request Interceptors: Any interceptor that also runs after a route has finished does so here. If one were timing the wrapped method, it could stop the timer here and append the runtime to the response payload.
8. Exception Filters: Catches any errors that make its way to the top of the call stack and returns them to the user. Even if the error is unrecognized, it will return a 500 response.
9. Response: The server sends the Express payload.

6.4.2 Registration

The website will have users tied together through an account entity, however most systems will operate in the context of a single user. In essence, authentication is fluid across any user within an account, but authorization is scoped to the permissions of an

individual user. Accounts have a primary user, or the adult user that created the account, and registration will not only create this primary user but also store their id as a foreign key in a new account entity.

It is worth mentioning that the default multi-tenant account design tries to minimize risk and provide a baseline that could be expanded upon in the future. For instance, the concept of a primary user is not directly tied into any logic aside from how the login information is stored and validated. This means it is possible that multiple adult users on an account could have separate logins that connect to the same account, however this does create more vulnerabilities as two or more sets of credentials increase the odds of access being leaked or used improperly. The login flow could also be adapted to include passwords for other users, or more emails, and so forth.

Figure 69 shows a sequence UML diagram of the registration process. A duplicate email will incur an error later into the process. The frontend is likely to either receive a 400 Bad Request if the initial request is malformed, or a 409 conflict error if the email is already taken. To prevent an adversary fishing for registered users, this method will be rate limited. Once the JWT token is sent, the user is logged in but their email is not flagged as validated yet and they will be unable to register to events until they do unless overridden by an admin.

A summation of the flow of user registration is as follows:

1. Send a POST request with the primary user information: email, password, name, and date of birth.
2. Validate if all required information is present, properly typed, and is contextually valid, e.g. are they old enough?
3. Store the hashed and salted password with the other user information in a new user entity.
4. Cascade-insert the primary user within a new account entity.
5. If successful, the emailing service will queue up a new email to be sent asking the user to confirm their email. This queue will retry this until the emailing API sends back an accepting response, the number of failures is too many.
6. Sign the JWT token with the account id and the primary user id and return it to the user so the frontend may store it.

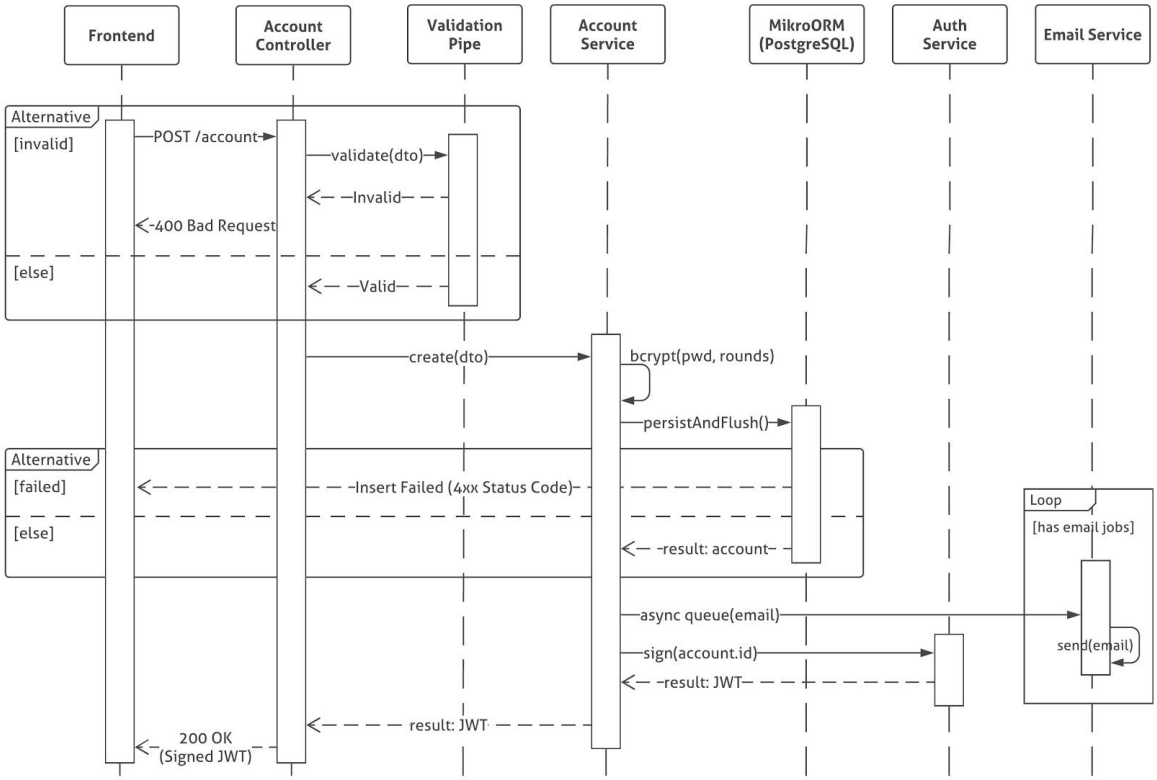


Figure 69: Account creation sequence diagram.

6.4.3 Login

The login flow utilizes the passport local strategy enforced by a guard on the login controller route. When the strategy is invoked, the request body is scanned for the email and password and passed through the auth service for verification. Once a user with the email is found and bcrypt determines if the password matches the hash, passport appends the user to the request object and returns to the controller. Figure 70 shows the sequence of methods called to verify the user's credentials and then create a token.

The controller then calls the auth service again, but the type of token returned depends on if there are multiple users on the account. Whenever there is only the primary user on an account, the auth service will grant a complete token. Whenever there are multiple users on the account, the auth service grants an incomplete token, requiring that the user proceed through the user selection flow to be granted a complete token. Registration also grants a complete token; the registration flow finishes by logging into an account with only one user.

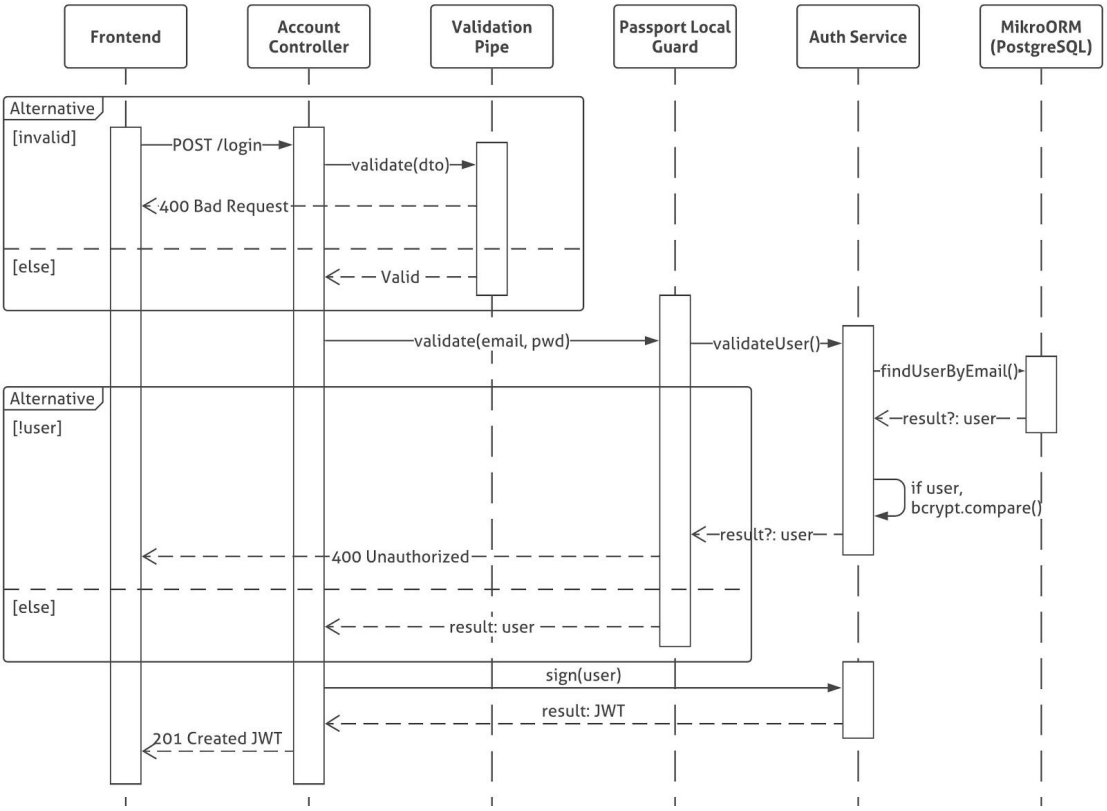


Figure 70: UML sequence diagram showing the backend login flow.

6.4.4 User Selection

If there are multiple users on an account, the login process requires a secondary step of selecting which user to represent on the website. The sequence of events for user selection is as follows:

1. Assuming the token decodes properly, or validating that the claims aren't tampered or that the token is expired, retrieve the claims.
2. If there is no account or user id claims, the token is malformed and rejected.
3. If there is a second expiration date within the claims, and the date has already occured, the token is incomplete and must be exchanged for another.
4. The presence of a user id that successfully is retrieved from the database means the token is complete. Append the user to the request object.
5. Otherwise, the presence of an account id means the token is incomplete, append the account to the request object.

The controller route responsible for switching users on an account utilizes a guard that first determines the validity of the token then allows the passport strategy to download

the account using the payload. The JWT strategy will use the account id if it's provided to find an account, otherwise it will find the account through the user relation, covering both incomplete and complete token types. Assuming the account is found, the controller method will have access to it and can call the auth service's switch method to generate a new JWT based on the user id provided. The JWT strategy will retrieve an account with all of its users populated, allowing the switch method to scan to ensure the user exists within the account before providing a valid token. This sequence of events is shown in Figure 71.

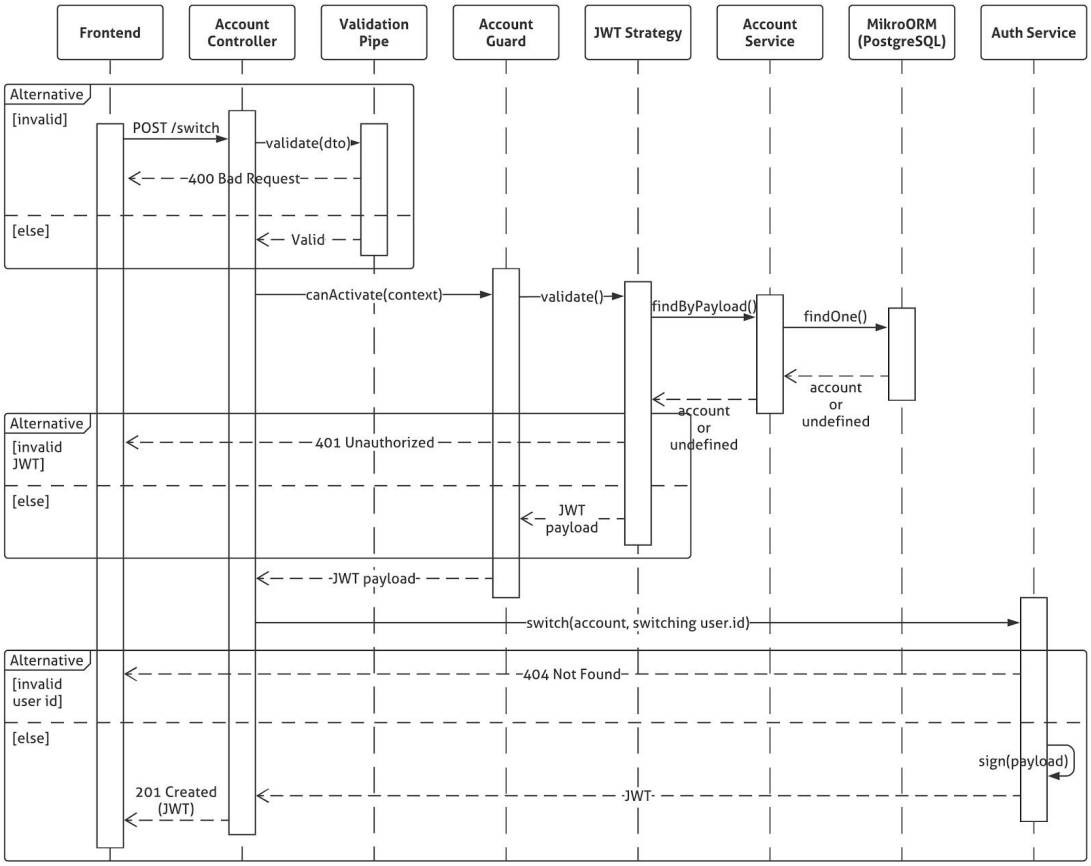


Figure 71: UML sequence diagram of the user switcher process.

6.4.5 Authorization

There will be three primary guards utilized throughout the backend, the account guard, user guard, and access control guard. The account guard and user guard reflect the token types and are used when only that base level of authorization is required.

- Account Guard: Only requires a valid JWT that is associated with an account. The account is retrieved and appended to the request object. This guard is only meant for use when switching accounts or invalidating all logged in users of the account.
- Use Guard: Requires a complete token associated with a user. The account is retrieved and then both it and the user claim within the token are stored in the request object. This guard is used for most endpoints requiring authentication and will be called prior to the access control guard.
- Access Control Guard: Utilizes other decorators that describe permissions metadata limiting the scope of the decorating method. For instance, a decorator could be used in conjunction with the access control guard that describes only users with update permissions for project entities may use the route. More custom decorators, such as an attribute decorator, may be used to limit permissions of users, e.g. by a min or max age attribute.

Where possible, methods will be generic and utilize the access control service from the Access Control package discussed in the authorization research section. The access control service will also power the access control guard. The guard will utilize metadata on each route and determine for all of the described roles if the user has the necessary permissions to do so. This is checked against all permissions as it is possible within the Access Control package to explicitly deny access to resources for a role.

In order to more easily use grants and multiple guards, a custom Auth decorator was made that can sequentially queue the user guard and the access control guard after having described the grant permissions for the route into one call. As seen in Figure 72, the grant describes that in order to use the update route which calls a project updating method the user must have the permissions to update any project resource. The access control guard retrieves the metadata that stores the grants from the Auth decorator and the roles of the user from the user guard iterates over them to call the permission method of the Access Control module, seen in Figure 73.

```
@Patch(':id')
@Auth({ resource: 'project', action: 'update', possession: 'any' })
update(
  @Param() { id }: FindProjectDTO,
  @Body() updateProjectDto: UpdateProjectDTO,
) {
  return this.projectService.update(id, updateProjectDto);
}
```

Figure 72: Example of a custom decorator, `@Auth()`, which combines the user guard and the access control guard and allows for any number of grants to be described.

```
// Does not use grants.some(...) because exclusive
// permission denying is possible.
const hasPermission = grants.every((grant) => {
  return this.ac.permission({ ...grant, role: roles }).granted;
});
```

Figure 73: Permission checking using the Access Control library by specifying grants in Figure 72.

6.4.6 CRUD Actions

The processes for the primary CRUD (create, read, update, and destroy) actions on entities are repeatable patterns that incorporate only minor changes. Some entities when created ask the email service to potentially notify users, such when events are created or updated. To avoid repetitive descriptions, CRUD actions will be explained in the context of the backend before the entity-specific behaviors are discussed.

6.4.6.1 Entity Creation

In RESTful APIs there are two primary means of creating entities, with POST or PUT requests. The difference between them lies in the necessity of idempotency, or that if a request is repeated multiple times the result will be the same outcome as if it were made once. PayPal is highly idempotent as servers making multiple requests should not implicitly repeat transactions if the buyer did not request it. PUT requests replace or update on subsequent requests. POST will create a separate entity for each time the method is called. Both of these philosophies have their uses, but given this website's need for a lot of protections to not accidentally destroy referential integrity between the entities, PUT is not in our design.

A typical creation sequence using the event entity as an example is modeled in Figure 74 with the code for the service method in Figure 75. For brevity, the authorization flow is condensed in the diagram. Create methods will have their own DTO for modeling which properties are required initially for the event to be made and then a MikroORM entity is created. The persist method then instructs MikroORM that the entity should be tracked, and the second parameter will flush, or save, the entity to the database.

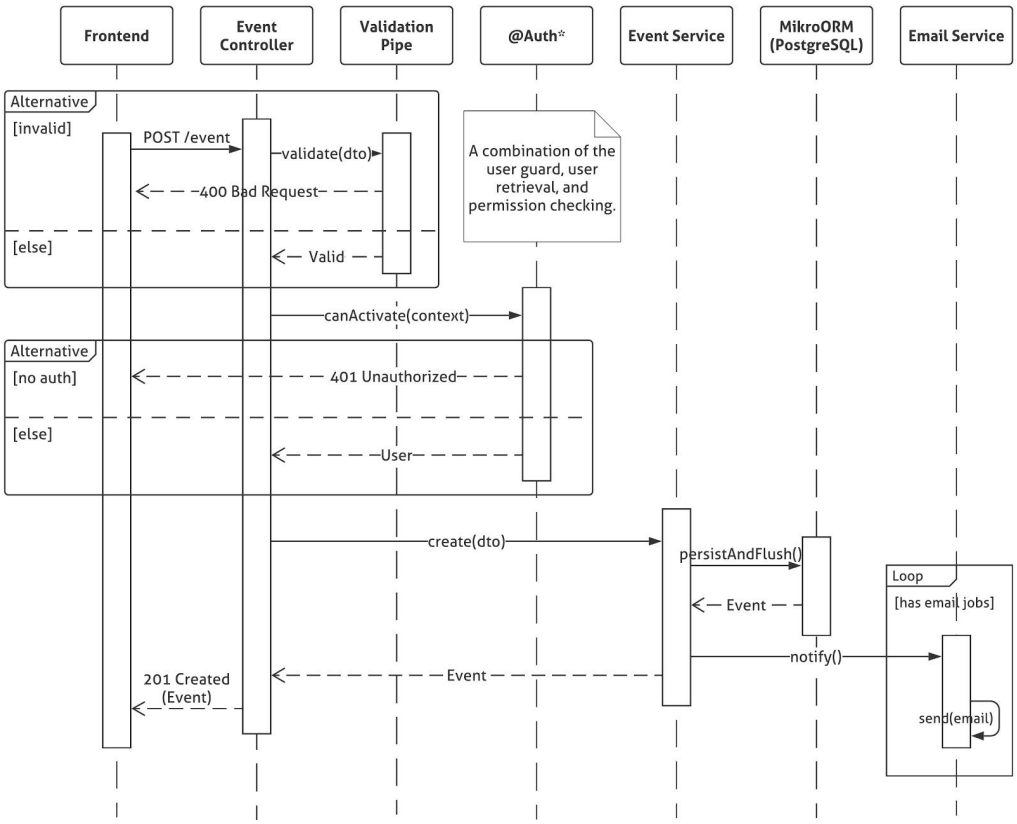


Figure 74: UML sequence diagram for event creation.

```
async create({ email, password, name, dob }: CreateAccountDto) {
  const account = new Account();
  const user = new User();

  user.email = email;
  user.email_verified = false;
  user.dob = dob;
  user.name = name;

  account.primary_user = user;
  account.users.add(user);
  account.password = await bcrypt.hash(password, 10);

  await this.accountRepository.persist(account, true);

  return account;
}
```

Figure 75: The create Event Service method for an account with the cascade-inserted primary user.

6.4.6.2 Entity Updating

Updating entities is done through the PATCH method. This method will utilize a DTO describing the properties the user is allowed to edit, as well as a DTO for finding the entity (typically just the entity's id). Updating methods will throw `404 Not Found` exceptions when the entity to be updated is nonexistent. A sequence diagram of this flow can be found in Figure 76. The diagram shows that after an event entity is updated, the emailing service is called. This specific type of notification will be sent to users who have already registered to this event if the times were changed. This may also include pertinent information such as the event location.

Updating entities will require contextually checking the metadata. Events in particular have many considerations around recurrence. For instance, modifying who may register to the event after registrations are already made may cause indeterminate behaviors on the website. The frontend will not want to show the event to this user, but the user is registered nonetheless and retrieving the registered events will indicate this. However, if the user wants to edit their registration, the event would be retrieved with permissions

incompatible with the user, and the user would not be allowed to interact with the event. This specific instance could be coded against, for instance by allowing incompatible users to interact with an event they're registered with (say, if an admin wants to override and create a registration). However this also has performance implications of requiring checks through joined data on the chance that someone is registered to an event they shouldn't be. Permission overrides such as this may lead to an uncaught situation, thus, updating events after registrations is a situation that should be avoided. Similarly, many entities have date ranges that the frontend will query against that are not referentially linked to other entities. If an event date is changed retroactively, payments, attendance, and volunteering entities may have incoherent data. In conclusion, updating is a core feature of the admin panel, however it is one that will require extensive testing and treaded carefully.

The code to update an account entity can be seen in Figure 77. Of note, MikroORM provides a method called `wrap()` that takes an entity as an argument and then the `assign()` method is called to update the wrapped entity by copying over the properties in the assign argument to the entity. This is not unlike the Object.assign() method in JavaScript, however the wrap method ensures only valid properties are copied into the entity. Then, the entity can be saved to the database with the repository `flush()` method. The key difference between `persistAndFlush()` (or synonymously `persist(entity, true)` as seen in the create code) and `flush()` is that entities retrieved from the database are already considered persisted. Persisting is instructing MikroORM that the entity should be managed, whereas flushing creates a transaction around all of the changes made to managed entities and makes the appropriate database queries.

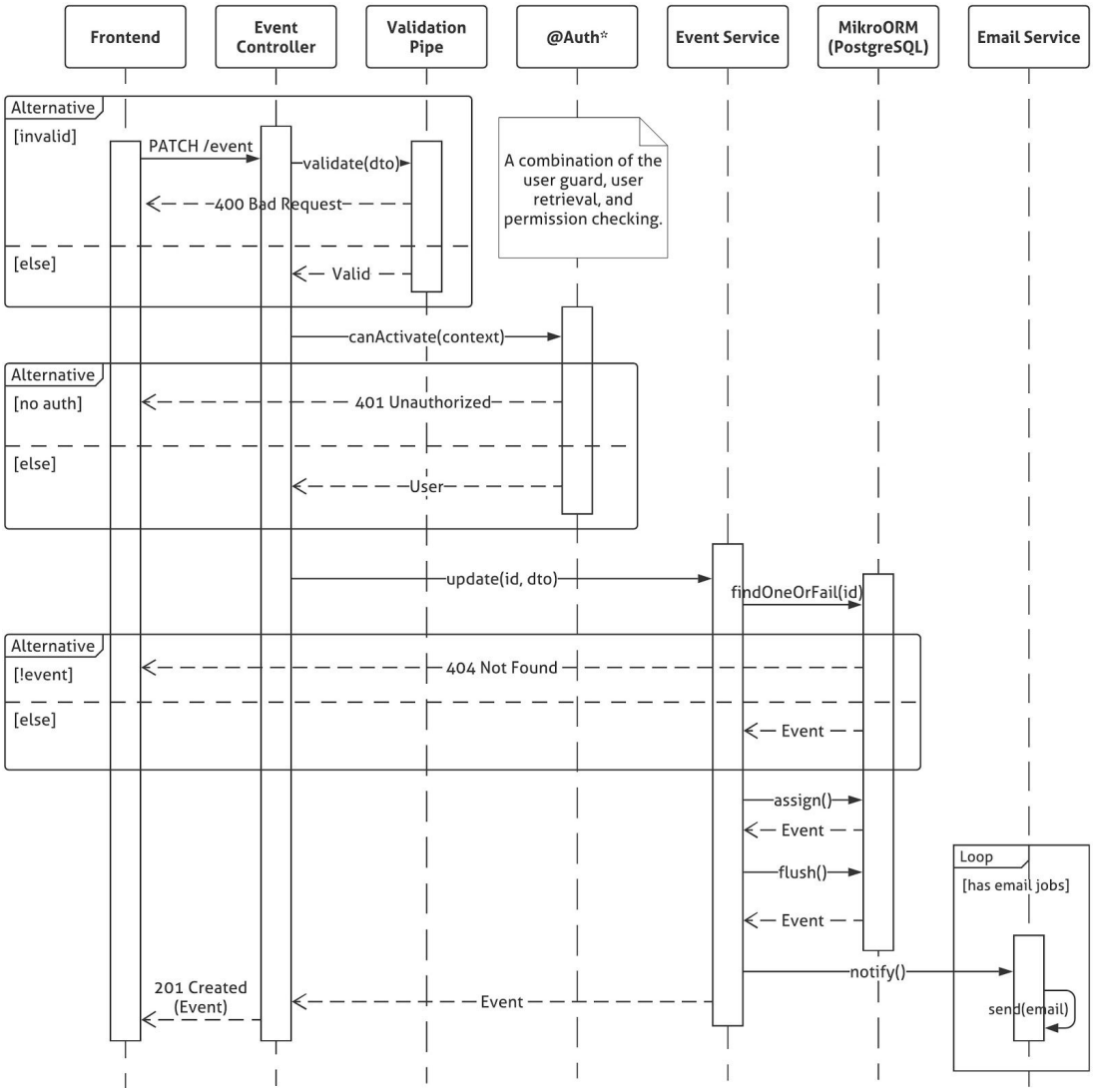


Figure 76: UML Sequence event for updating event entities.

```
async update(id: number, updateAccountDto: UpdateAccountDto) {
  const account = await this.accountRepository.findOneOrFail(id);

  wrap(account).assign(updateAccountDto);

  await this.accountRepository.flush();

  return account;
}
```

Figure 77: Updating method example for account entities.

6.4.6.3 Entity Reading

Reading entities from the database can either be a step in a service method, or the desired action itself. Since the concept is straightforward, a sequence diagram is not included. For reference, the flow is not different from that in Figure 74.

When finding a singular entity, a findOne() or findOneOrFail() method can be used. Most methods will use the findOneOrFail() method as it is used to quickly exit from the current call stack because the entity or related entities it was tasked to find do not exist and thus no actions can be taken on them. However, some methods should continue when the entity was not found, in particular authentication. When a user is not found, immediately telling the requester that such a user doesn't exist is a bad practice in terms of security. While non-admins will not be able to retrieve users that aren't within their account, an example where this occurs is the login flow. Before checking the validity of a password, a user is looked up via the email. The flow can immediately exist, however if it told the user that the email doesn't exist, it lets the user retry to try to fish for existing users until they get an invalid password request. Authentication thus does not throw immediately when a user is not found and casts all issues during this flow to `401 Unauthorized` exceptions.

Some expressions of entities require pagination, primarily the admin panel. While there may only ever be a couple hundred users, there may be thousands of events and a multiple of that many event registrations. Asking the backend for every entity will slowly become a performance issue for the backend and for the device retrieving them. Thus, pagination will be used on all tables with configurable defaults. An example of pagination using MikroORM is shown in Figure 78. The method takes a limit, or the total number of entities to retrieve, and an offset, the total number of entities to skip. The findAndCount() method will create a query that skips offset-many rows and takes the following rows up to the limit. The structure of the response is split into an array for the entities, which can be empty, and a number denoting the total number of this entity type. This allows for the frontend to divide the total number of entities by the limit and separate the data into that many pages. When the next page is desired, the offset is increased by the limit number.

```
findAll(limit: number = 20, offset: number = 0) {
  return this.accountRepository.findAndCount({}, { limit, offset });
}
```

Figure 78: An example `findAll()` method with pagination for accounts.

6.4.6.4 Entity Deletion

The backend will utilize a few modes of deletion to accommodate the related entity structures. Few entities when removed will physically destroy them from the database. One form of deletion alternative that will be utilized is a process called soft deletion. A flag is set on the entity that denotes that the entity is no longer considered in circulation. Users on the frontend would not see the entity anymore. For example, if an admin no longer wants a project job to be in circulation they will delete it. However, the entity can not be truly deleted if any volunteers completed previous work for the job as then the description of the job is lost. Here, an admin would be soft-deleting the job and afterwards volunteers registering for events would not see the deleted job in the selection list.

Due to the highly connected nature of the application, few entities can truly be removed without a dramatic loss of data. If an event were deleted, the registrations would also need to be removed, and thus so would the attendance data. Payments and jobs completed can hold their own when their associations are removed as the information they hold is still valuable even if the context is lost (invoice numbers and work hours). Even then, it will be necessary to truly delete entities and information. It is not an unreasonable request for a parent to want their account deleted from the application, so when accounts or users are deleted, cascades will remove all associated entities for that user.

6.4.7 Uploading Files

Files can only be uploaded to the backend through the `multipart/form-data` encoding, a dictionary (key-value) structure that is not directly usable by Nest.js without conversion. Multer, the library which will be accepting files, parses the multipart encoding and outputs a simple object for a JSON body representation that can then be run through the validation pipe. Multer is used by an abstracted Nest.js interceptor seen in Figure 79 that can be configured to either accept any files, an array of field keys, or a

singular field key. If the field key is invalid, or a limit for the file size specified is exceeded, Multer will reject the input and throw an error. Otherwise, Multer will generate the body object and the file or files and attach it to the request. If the uploaded file is a new free lunch form, admins with this notification type set will be sent an email. Then when an upload is deleted the native file system can be used asynchronously using the `fs.unlink()` function. This process is outlined in a sequence diagram in Figure 80.

```
@Post('upload')
@UseInterceptors(FileInterceptor('reduced_lunch', {
  limits: {
    fileSize: 15000000
  }
}))
upload(@Body() uploadFileDto: UploadFileDTO, @UploadedFile() file) {
  return this.uploadService(uploadFileDto, file);
}
```

Figure 79: FileInterceptor for a single field with a 15MB file limit.

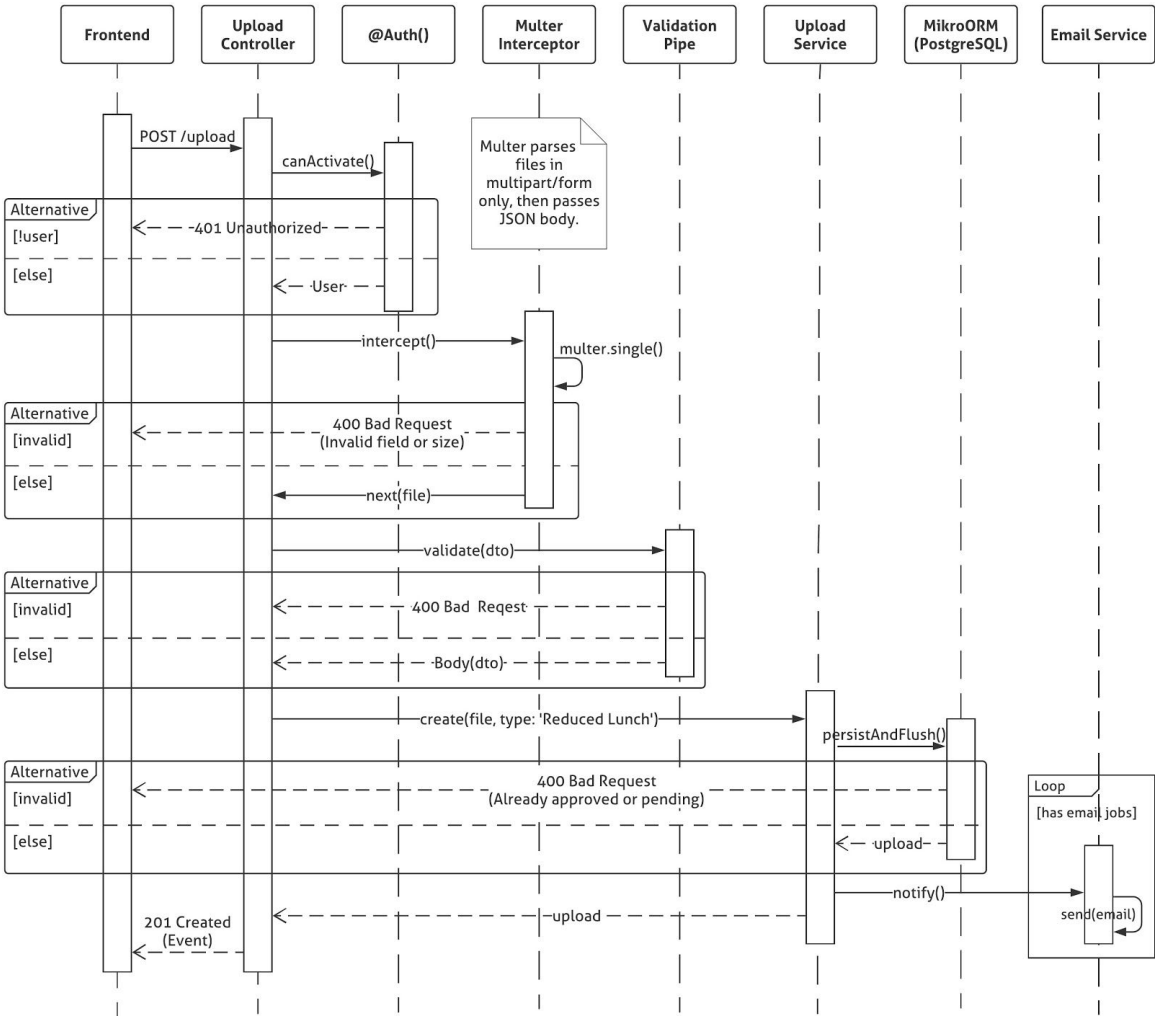


Figure 80: UML sequence diagram for the file uploading flow.

6.4.8 Attendance

When a user visits an event and has their attendance taken, steps are taken to tie together registrations and volunteer work. The website needs to be approachable to all users, especially those who just registered and would prefer to interact with a person instead of standing in the corner of the room trying to register on the phone. When a parent visits an event, they will have the opportunity to pay in person (if permitted by the event and OMC, unless they have their fees waived) and the admin can then start the flow for creating an attendance record by retrieving the person, ideally by email, but name searching will also be supported.

Once the primary user is found for the account, the admin will be able to register the user by selecting the students for the event that the parent has entered after registration. Usability tests will need to be performed to determine if the step that requires parents to enter their children into the application is not confusing or is too disconnected from the registration step. It would be difficult to register without having the student information in the website, however a compromise may be possible by allowing the admins to also immediately create users for those recently registered.

We would not want too much of this work offloaded onto the person taking attendance, but processes may need to be taken to determine how users who wish to attend events but aren't interested in using the website to register themselves should be handled. Nonetheless, the sequence diagram seen in Figure 81 shows how the different services are conditionally involved depending on if a registration is not already present. Volunteers will also have their job created at this point as unless the job is denoted as being remotely handled when it was created or will need to be submitted manually by the volunteer for approval.

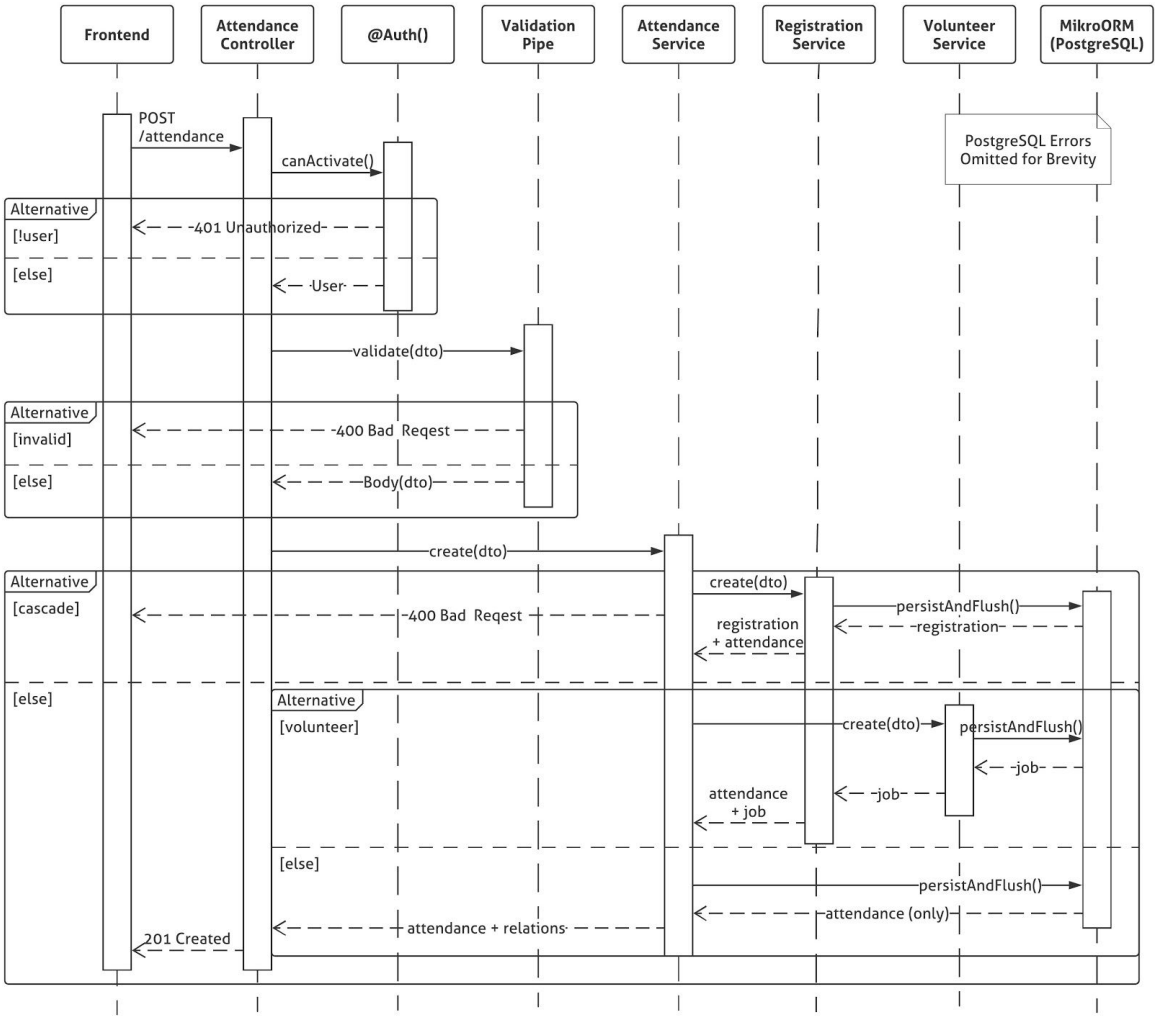


Figure 81: UML Sequence diagram for creating and cascading the various attendance related entities.

6.5 Networking

The website will require a root domain for the frontend and a subdomain for the backend. Alternatively, two subdomains can be used if OMC does not wish to pay for the second domain and have to manage it, however, special mailing records will also need to be installed on this domain and as such, the team will need access to the either the dashboard which has the current domain records, or the domain's nameservers be pointed to either AWS, DigitalOcean, or other provider of OMC's choosing and managed alongside the VM.

Once the domain is handled, an address record will need to direct the root and subdomains to the IP of the server instance. Once the DNS propagation is successful and the domain is pointing to the server, NGINX will be installed to proxy the requests to the frontend or backend where appropriate. The frontend will receive traffic from the WWW and root domains (if not meant to go to a subdomain), whereas the backend will receive traffic from the API subdomain, e.g. `api.omc-events.org`. In order to remain secure, non-HTTPs traffic will be enforced and non-HTTPs traffic will be upgraded using a permanent redirect to coax the browser into not making an unauthenticated request a second time. The only exception where traffic does not need to be encrypted is when the frontend is making a server-side request on the machine to the backend. In order to use SSL, the automated tool Certbot is used and the appropriate domains and subdomains are listed and configured automatically. Certbot will generate the private and public keys, configure NGINX to utilize them, and validate the ownership of the domain before being validated by their certificate authority [34].

Documentation will be included on how LetsEncrypt and Certbot work and how the NGINX configuration files for the domains operate. The code that will handle the proxy for both the frontend and backend servers is shown in Figure 82. For example, the client should never remove the configuration location that allows LetsEncrypt to their acme challenge files from the server. LetsEncrypt certificates expire every 90 days and are typically renewed at the 60 day mark with the server retrying until it does so. The website would give an ominous warning to mobile users if the certificate expired as well as make it impossible to make PayPal payments.

```
location / {
  expires $expires;

  proxy_redirect                    off;
  proxy_set_header Host             $host;
  proxy_set_header X-Real-IP        $remote_addr;
  proxy_set_header X-Forwarded-For  $proxy_add_x_forwarded_for;
  proxy_set_header X-Forwarded-Proto $scheme;
  proxy_read_timeout                1m;
  proxy_connect_timeout             1m;
  proxy_pass                        http://127.0.0.1:3030;
}
```

Figure 82: NGINX site configuration at the root of a domain or subdomain to proxy requests to the port specified under proxy_pass.

NGINX will be configured to gzip compression and will be set to cache images, however since both the frontend and backend will sit behind a proxy and utilize specialized server delivery mechanisms, direct HTTP2 header pushing is not possible. Similarly, the assets during compilation change, making it not ideal to manually add files to be pushed to the user. However, the uploaded files will likely be aliased to some static location that overrides the frontend proxy, such as `/uploads`, with files only being referenced by a unique name and not indexed.

6.6 Server Management & Tooling

The server components can be run using the PM2 library [35]. Typically, the frontend and backend are run using the npm commands, but even though both will components will be designed to be highly tolerant to failures and errors, things such as unexpected power outages and other server restarts may cause the website to close. PM2 is a library that can install npm commands as aliased tasks that can be configured to automatically restart when they fail or the code needs to be recompiled.

Documentation will be provided to the clients on how to utilize and manage the PM2 library, as well as how it can be configured to increase the concurrency of the application if the server has the appropriate resources.

6.7 Gantt Chart

We have been utilizing a Gantt chart to model our current progress and what still needs to be done, seen in Figure 83.

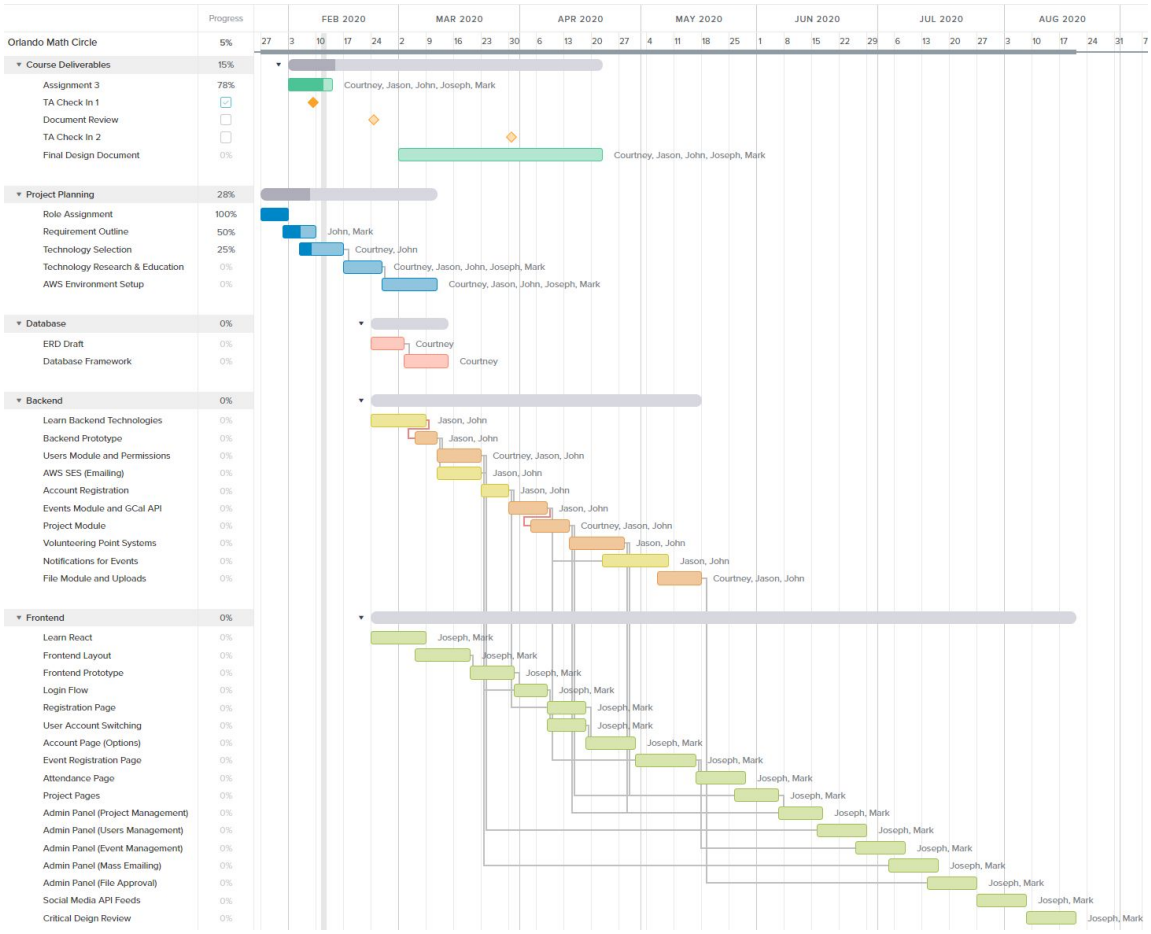


Figure 83: Gantt chart used for outlining the project sequence.

6.8 Use Case

This use case diagram in Figure 84 demonstrates the actions users can take once they have logged in to the web page, or set up an account for themselves. The clients specified that they wanted accounts set up in a very specific fashion. Accounts would be broken down into a few different categories, those being adults, children, and volunteers. Only users who are above the age of 13 will be able to sign up for their own accounts, due to COPPA regulations. The main account type will be adults who want to send their children to events. Here is an explanation of the actions users can take:

- Adults will be able to add children to their account, and give each child a profile under the adult's login
- The adult account can view the event calendar to see what events are coming up
    - The adult account can add their children to an event, so that their children can attend

    - The adult can sign up to volunteer for an event
- Adult accounts can sign in themselves and any children attached to their account to an event through an attendance sheet once the event has begun
- Adults can upload files to the administrators of the website, such as forms proving a child is eligible for free/reduced lunch
- Adult accounts can view the volunteer handbook to review the rules volunteers must follow
- Admin accounts can create a new event on the event calendar
- Admin accounts can view the attendance sheet of an event and see all of the members who attended
- Admin accounts can edit user accounts if they need to

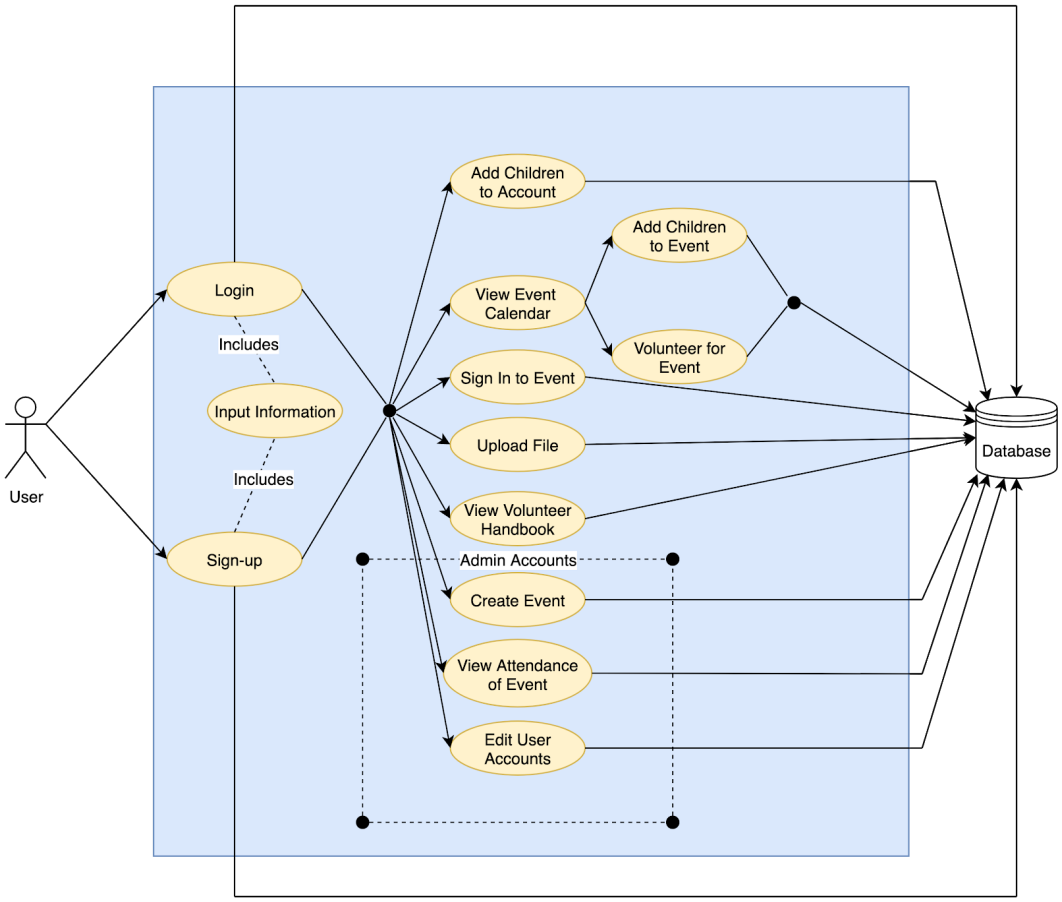


Figure 84: Use case diagram for user interactions

7. Budget, Financing, and Resources

Amazon AWS is the development solution provider for our project. A 12-month credit will be used at the beginning of the project and will allow for the development of all website features without incurring any charges. Certain decisions in the utilization in the technology will have impacts on the production expenses accrued from the website. A detailed analysis of the two recommended hosting providers, DigitalOcean and AWS, are discussed in the Hosting Providers section of this document.

- The application will be developed on an Amazon EC2 Micro (t2.micro) Ubuntu server instance.
- If DigitalOcean is used, costs will be $15 monthly with some change depending on the size of the backups and the email volume. If AWS is used, costs depend heavily on the design implementation, but are likely to be $17 monthly, but less than $20.
- AWS SES costs are effectively free if an AWS EC2 server is used to host the website. If DigitalOcean is used to host the website, emails will be $0.10 per 1,000 emails sent.
- The database can either be managed or hosted directly on the EC2 instance, though the scalability of the application can be greatly improved by doing the former, but will have cost considerations for the client in the future. Currently, either solution is free using credits for the project.
- The domain for the website will need to be purchased by our clients and records routed to the VM's address. Domains are typically less than $15 yearly for the usual top-level domains.
- Graphic design resources will be provided by one of our clients, Sheina Rodriguez.

8. Project Plan

As a team we have agreed upon the following plans and procedures for the building phase of the website.

8.1 Build

This current semester (Spring 2020) we have focused our development on the basic support structure of the website. The frontend currently has UI mockups, operational prototypes, and mock data for all of the pages and the backend has fully operational authentication, authorization, account switching, entity schemas are synced to the AWS PostgreSQL database, and each entity has operational CRUD actions.

As we transition into the summer months, development efforts will shift to setting up external APIs such as PayPal and AWS SES and creating the specialized methods for handling events, attendance, and payments. With the CRUD actions already created, half of the team will also begin working on the admin panel while working with Sheina to build frontend design mockups and prototypes. By the start of the Fall 2020 semester, the backend will have all of the core features operational through the REST API, backend end-to-end tests written for continuous integration on GitHub, and operational frontend prototypes for each core feature.

Throughout the Fall 2020 semester features will be demoed to both the clients, other OMC faculty, and the Senior Design professors. It is at this point finer details of the application such as email templates, developer documentation, interaction testing, frontend end-to-end tests, and concurrency tests will be performed. We want this phase of development to be around polishing the UI around an already functional and satisfactory app and implementing other features the clients were considering if time permitted. It is at this stage that the administrative tasks such as applying for SES emailing limits to be unrestricted, the PayPal developer app created, and the domain purchased.

8.2 Prototype

The website can be constructed locally on each developer's machines, allowing us to utilize the AWS EC2 server for hosting prototypes for testing on mobile devices, usability tests, and get design reviews from OMC personnel. As a prototyped feature of the website is constructed and committed to the GitHub repository, a Git hook will be

executed on the server to automatically pull the changes, recompile the application, and restart the website. A sufficiently complete prototype may be used at certain events to garner opinion from parents about how they feel about the application, and OMC staff about the attendance process. Functionality aside, the design is important in ensuring that it feels natural to use and that both event attendees and OMC personnel want to use the website.

8.3 Test

The backend will undergo rigorous end-to-end tests with the AVA Node.js test runner library. Continuous integration on GitHub will be used to run tests on a seeded docker PostgreSQL database. Each controller will be tested to compare specific inputs to expected outputs and randomized inputs, such as with recurrence rules, to ensure conversions are adequate. The features or processes of notable importance for testing include:

- Ensuring higher-level methods and routes are not accessible to the wrong roles.
- Comparing the requests that generate events to the returned times to ensure the database is properly generating dates from the recurrence rules, and that recurrence rules generate the proper dates.
- Check the date timezones to ensure at no point is a non-UTC date being sent to the database, or that a localized date from the browser is not being converted to a proper UTC date.
- Recurring events will also be tested to ensure parity with Google Calendar behaviors during event modifications

The frontend will also include end-to-end tests, however the primary testing mechanisms will be through usability and concurrence tests. Each feature will need to be accomplished on the various mobile browser types, versions, and errors forced to each route to ensure the frontend guards itself against unexpected errors, primarily networking or slow server responses due to latency.

8.4 Evaluation

With the last step of our build plan, we are going to work directly with our clients at Orlando Math Circle, their boss, and likely other staff or event attendees to finalize our product for them. The purpose of this step is to make sure that we are not only creating a website that fulfills our requirements for Senior Design, but also is something that we can

feel confident handing over to our sponsors that solves the problems they came to us with. We will be taking the feedback that we get from each evaluation to adjust the website so that the function and design are best suited for the client.

In order to save us from getting stuck with a lot of changes or updates at the last second, we will make sure that they are sufficiently happy with a working prototype with only the core requirements. Feedback will be gathered as the clients interact with the website, ask questions, and describe any modifications or concerns they have with any of the features. It is important that once the calendaring and emailing features are functional we get feedback on how it operates to ensure we don't graft features onto a set of core functionalities the clients are not happy with. We intend to have the website completed with enough time for the clients to test the website on their own at events and develop further opinions on it as it's being tried so a final round of modifications can be made before we hand it off to them.

Evaluation is important to us and to our clients so we can avoid a repeat of the failures of their past solution. If we end up having to backtrack on features then it is less likely that the clients can receive the stretch goals. Similarly, none of us have experience developing applications with children in mind, or attempting to make a native mobile experience in general. The concepts used by this website will be simple, but much of their success will ride on how they are received by the users who will ultimately be registering for events.

9. Milestones

Est. Completion	Milestone	Status
2/7	TA Check-In 1	Completed
2/12	Project Requirements / Initial Design Document	Completed
2/28	Project Status and Document Review	Completed
3/7	Hardware and Database Setup	Completed
3/30	API and Database Prototype	Completed
3/30	TA Check-In 2	Completed
4/21	Final Design Document	Completed
5/10	Frontend Mockup with Client	In Progress
6/31	Frontend Prototype	In Progress
10/30	Project Demo for Professors	No Progress
11/15	Senior Design Conference Paper	No Progress
11/25	CECS Senior Design Showcase	No Progress
11/30	Final Project Presentation	No Progress
11/30	Final Project Document	No Progress

10. Project Summary & Conclusion

As shown in our design plan, we want to make a website that will create a user friendly experience for students, parents, and volunteers. The design will promote student engagement and attendance by eliminating communication issues through an interactive platform. Students who attend Orlando Math Circle events will feel more engaged through the loyalty points system that encourages event participation. They will also have the opportunity to develop their computer science skills through hands on web development on the website that is built. Parents will never have to worry about missing an event since they will be able to see and sign up for all the updated events. They will always have all the information needed to attend these events. Our design of the volunteer accounts will encourage people to help by making it easier to become a volunteer. Our website also makes volunteer hours easier to track and verify. We believe that our website will help to encourage event turnout and student engagement while minimizing any miscommunication. We hope that our website will provide Orlando Math Circle with a cost effective and user friendly platform that they will be able to use and build on for years to come.

Although we have already completed a great deal of research in the making of our design, we will not be limited to the design we have laid out. We will continue to research and develop our design as we implement it. This document covers many of the technical and legal details about our design and build plan. This includes hosting, emailing, legal issues, security, budget/financing, frontend, backend, and database technologies. We are excited to begin implementing our design and look forward to learning from the experience.

11. References

[1] DigitalOcean. (2019). Pricing on DigitalOcean - Cloud machine & storage pricing. Retrieved from https://www.digitalocean.com/pricing/

[2] Amazon Web Services. (n.d.). Simple Monthly Calculator. Retrieved April 1, 2020, from https://calculator.s3.amazonaws.com/index.html

[3] VPS Benchmarks. (2020, March 6). DigitalOcean vs Amazon EC2. Retrieved from https://www.vpsbenchmarks.com/compare/docean_vs_ec2

[4] Amazon Web Services. (2020). Amazon EC2 Pricing. Retrieved from https://aws.amazon.com/ec2/pricing/

[5] Amazon Web Services. (n.d.). CPU Credits and Baseline Performance for Burstable Performance Instances. Retrieved April 1, 2020, from https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/burstable-credits-baseline-concepts.html

[6] GitHub. (2019). Role and Attribute based Access Control for Node.js. Retrieved from https://github.com/onury/accesscontrol#readme

[7] mongoDB. (n.d.). NoSQL Databases Explained. Retrieved February 25, 2020, from https://www.mongodb.com/nosql-explained

[8] Oracle. (2020). What is a Relational Database? Retrieved from https://www.oracle.com/database/what-is-a-relational-database/

[9] Hristozov, K. (2019, July 19). MySQL vs PostgreSQL -- Choose the Right Database for Your Project. Retrieved from https://developer.okta.com/blog/2019/07/19/mysql-vs-postgres

[10] PostgreSQL Tutorial. (n.d.). PostgreSQL vs. MySQL. Retrieved from https://www.postgresqltutorial.com/postgresql-vs-mysql/

[11] Node.js v14.0.0 Documentation. (n.d.). Retrieved from https://nodejs.org/api/

[12] Documentation | NestJS - A progressive Node.js framework. (n.d.). Retrieved from https://docs.nestjs.com/

[13] Express - Node.js web application framework. (2017). Retrieved from https://expressjs.com/

[14] tc39/proposal-decorators: Decorators for ES6 classes. (2019, September 22). Retrieved from https://github.com/tc39/proposal-decorators

[15] Dependency Injection Providers. (n.d.). Retrieved from https://angular.io/guide/dependency-injection-providers

[16] HTTP response status codes. (2020, February 23). Retrieved from https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

[17] Middleware | NestJS - A progressive Node.js framework. (n.d.). Retrieved from https://docs.nestjs.com/middleware
[18] Interceptors | NestJS - A progressive Node.js framework. (n.d.). Retrieved from https://docs.nestjs.com/interceptors
[19] Pipes | NestJS - A progressive Node.js framework. (n.d.). Retrieved from https://docs.nestjs.com/pipes
[20] Guards | NestJS - A progressive Node.js framework. (n.d.). Retrieved from https://docs.nestjs.com/guards
[21] Validation | NestJS - A progressive Node.js framework. (n.d.). Retrieved from https://docs.nestjs.com/techniques/validation
[22] typestack/class-validator: Validation made easy using TypeScript decorators. Retrieved from https://github.com/typestack/class-validator
[23] expressjs/multer: Node.js middleware for handling `multipart/form-data`. Retrieved from https://github.com/expressjs/multer
[24] Knex.js - A SQL Query Builder for Javascript. (2020, April 16). Retrieved from http://knexjs.org/
[25] jaredhanson/passport: Simple, unobtrusive authentication for Node.js. Retrieved from https://github.com/jaredhanson/passport
[26] OptimalBits/bull: Premium Queue package for handling distributed jobs and messages in NodeJS. Retrieved from https://github.com/OptimalBits/bull
[27] Socket.io. (n.d.). Socket.io: Broadcasting. Retrieved April 1, 2020, from https://socket.io/get-started/chat/#Broadcasting
[28] Getting Started - React. (n.d.). Retrieved from https://reactjs.org/docs/getting-started.html
[29] Figma. (n.d.). Retrieved February 29, 2020, from https://www.figma.com/
[30] Vue.js. (n.d.). Vue.js API: Directives. Retrieved from https://vuejs.org/v2/api/#Directives on April 2nd, 2020.
[31] Alexchopin. (n.d.). Nuxt.js - The Vue.js Framework. Retrieved from https://nuxtjs.org/
[32] IETF. (2009, September). Internet Calendaring and Scheduling Core Object Specification (iCalendar). Retrieved from https://tools.ietf.org/html/rfc5545
[33] REST API Tutorial. (n.d.). Retrieved from https://restfulapi.net/
[34] Certbot. (n.d.). Certbot Instructions. Retrieved from https://certbot.eff.org/instructions on April 10, 2020.
[35] PM2. (n.d.). PM2 Startup Script Generator. Retrieved on Monday, April 13, 2020, from https://pm2.keymetrics.io/

[36] Amazon Web Services. (2019, July 12). What is Amazon SES? Retrieved from https://docs.aws.amazon.com/ses/latest/DeveloperGuide/Welcome.html

[37] AWS. (2019). Complying with DMARC using Amazon SES. Retrieved from https://docs.aws.amazon.com/ses/latest/DeveloperGuide/send-email-authentication-dmarc.html

[38] OpenJS Foundation. (2020, February 11). Node.js v12.16.1 Documentation. Retrieved from https://nodejs.org/api/crypto.html

[39] OWASP. (2020, March 22). Cross Site Request Forgery (CSRF). Retrieved from https://owasp.org/www-community/attacks/csrf

[40] Dan Arias. (2018, May 31). Hashing in Action: Understanding bcrypt. Retrieved from https://auth0.com/blog/hashing-in-action-understanding-bcrypt/