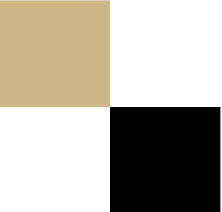
BoilerTime

CS 30700

Design Document

Team 22:

*Mahad Faruqi*

*Justin Lai*

*Henry Mayer*

*Nicholas Powers*

*Zachary Sy*

*Joshua Yang*

Index

1. Purpose
   1. Functional Requirements
   2. Non-functional Requirements
2. Design Outline
   1. High Level Architecture Overview
3. Design Issues
   1. Functional Issues
   2. Non-functional Issues
4. Design Details
   1. Class Diagram
   2. Service Execution Diagram
   3. UI Mockups
   4. Database Diagram
   5. HTTP Requests

# Purpose

Students at Purdue University struggle to find the optimal schedule that balances the various factors of their day. BoilerTime eases this by ingesting user preferences and matching it with a variety of data sources to find the optimal class sessions based on student feedback, distance between buildings, bus times, previous grade distributions, and friends’ schedules. As students already tend to closely reference individual services separately such as Rate My Professors, Google Maps, and BoilerGrades during class registration, this service combines them all into one application interface and provides a detailed analysis to recommend a complete schedule.

## **Functional Requirements**

1. As a student, I would like to register for a BoilerTime account using my *@purdue.edu* email.
2. As a guest, I would like to be able to use limited features without providing my *@purdue.edu* email.
3. As a user, I would like to ensure my email address is verified before using the website.
4. As a student, I would like to be able to reset my password if I forget my login details.
5. As a student, I would like to be able to create a profile that contains my name, profile picture, year at Purdue, and other information about me
6. As a student, I would like to input my class choices and categorize them into required/major and elective/flex/interest classes.
7. As a student, I would like to be presented with a filtering by tags or categories feature for the classes.
8. As a student, I would like to be presented with a live search by keywords feature for the classes.
9. As a student, I would like to be notified if my required/major classes outright do not work together.
10. As a student, I would like my schedule to contain all of my required class choices.
11. As a student, I would like classes that are not already full.
12. As a student, I would like to bookmark certain classes.
13. As a student, I would like to be notified through email if a slot opens in one of the classes I hoped to take.
14. As a student, I would like to see a calendar view of my schedule.
15. As a student, I would like to see a list view of my schedule.
16. As a student, I would like to be able to find a schedule that considers all of my preferences in one.
17. As a student, I would like to be able to prioritize my classes based on the time of day.
18. As a student, I would like to see the past average GPAs of classes based professors.
19. As a student, I would like to see the peer ratings of each of my professors.
20. As a student, I would like to see the route between each of my classes.
21. As a student, I would like to create a schedule that prioritizes minimum walking distance.
22. As a student, I would like to create a schedule that prioritizes overall professor rating.
23. As a student, I would like to be able to specify my residence hall or bus route.
24. As a student, I would like to plan around other meetings, or TA sessions outside of my Purdue class schedule.
25. As a student, I would like to be able to see rankings of different schedule options.
26. As a student, I would like to export my schedule to external calendar apps.
27. As a student, I would like to be able to export my schedule as a photo or PDF.
28. As a student, I would like to be able to interact with my schedule on the calendar.
29. As a student, I would like to see the class distances visually through a map with locations of classes marked.
30. As a student, I would like to be able to view classroom ratings and teaching assistants.
31. As a student, I would like to be able to write classroom ratings based on the convenience of access, quality of seating, and availability of technology in the classroom
32. As a user, I would like to be able to rate a class, based on the strictness of prerequisite requirements, the pace of materials covered, the depth of the materials covered,
33. As a student, I would like to be able to edit and delete my past ratings.
34. As a student, I would like to be able to flag and report malicious ratings.
35. As a student, I would like to be able to write ratings of teaching assistants based on helpfulness of answering questions, responsiveness, and fairness of grading.
36. As a friend, I would like to join a group so that I can have classes with friends.
37. As a user, I would like an easy way to create a group for my friends through unique links.
38. As a friend, I would like my group’s schedule to match with mine for certain classes.
39. As a friend, I would like to be able to see all of my friend’s schedules on one screen.
40. As a user, I would like to be able to share a created BoilerTime group .
41. As a guest, I would like to have my schedule saved on my browser, even if I am not logged in.
42. As a user, I would like to be able to find other BoilerTime users in my classes if they consent (i.e. the “discover” functionality on social media platforms).
43. As a user, I would like to see an introduction page to the website so that I can learn about the special functionalities of the website.
44. As a user, I would like to be able to access my collected data so that I can track my digital footprint.
45. As a user, I would like to be able to change my password from my account settings.
46. As a user, I would like to be able to customize the accent and themes of the website.
47. As a user, I would like to be able to have two-factor authentication on my account.
48. As a user, I would like to be able to create unique links to share my BoilerTime schedule.
49. As a professor, I would like to be able to see feedback from my students.
50. As a user, I would like to be able to provide feedback on my generated schedule and errors that I see in it.
51. As a student at another university, I would like to be able to port the BoilerTime service to my university and host it on my own infrastructure.
52. As a user, I would like to see recommended classes that people have taken together.
53. As a user, I would like to see a list of trending and popular classes.
54. As a user, I would like to import and view my previous schedules in my profile.
55. As a user, I would like a limited functionality version of the platform for use on a mobile or smaller-screen device.
56. As a user, I would like to be able to set preconditions of times that I need to reserve, like religious prayers (ex. Friday Prayers for Muslims has two times on Friday).

## **Non-Functional Requirements**

### Architecture and Performance

1. A Java algorithm that analyzes schedule options to create recommendations
2. A Vue.js front end to dynamically render a data-driven user interface on the client side
3. A middleware built on Node.js to handle request-response cycles and dependencies
4. Integration of a cloud-based analytics tool to understand user behavior
5. Separation of the front end and back end for modularity and maintainability
6. Data storage and retrieval through various API endpoints
7. Use of caching strategies to minimize redundant requests
8. Use of load-balancing strategies to ensure support for concurrent requests and at least ~1,000 simultaneous users
9. Use of optimization strategies to ensure average initial website load time across devices is less than 5 seconds
10. Use of prerendering strategies to ensure time to first byte when navigating across functionality is optimized for network and device conditions

### Security

1. Implementation of rate limiting to prevent excessive or malicious API requests
2. Requiring CAPTCHA verification for guest users
3. Automated logging and flagging of user requests and session details (i.e. user agents, IP addresses)
4. Compliance with data privacy regulations
5. Requiring a client generated JSON web token (JWT) to access the back end’s API endpoints
6. Ensuring that user data is encrypted both in transit and at rest

### Hosting/Deployment

1. Deployment of the back end to a cloud infrastructure for close to 24/7 end-user availability and easy future scalability
2. Integration of a cloud-based monitoring tool for resource utilization and uptime to allow quick response to performance or security issues
3. Regular backups of user data to protect against data loss and corruption
4. Use of version control to manage the platform’s code changes
5. Proper configuration of a firewall on the back end to restrict access

### Usability

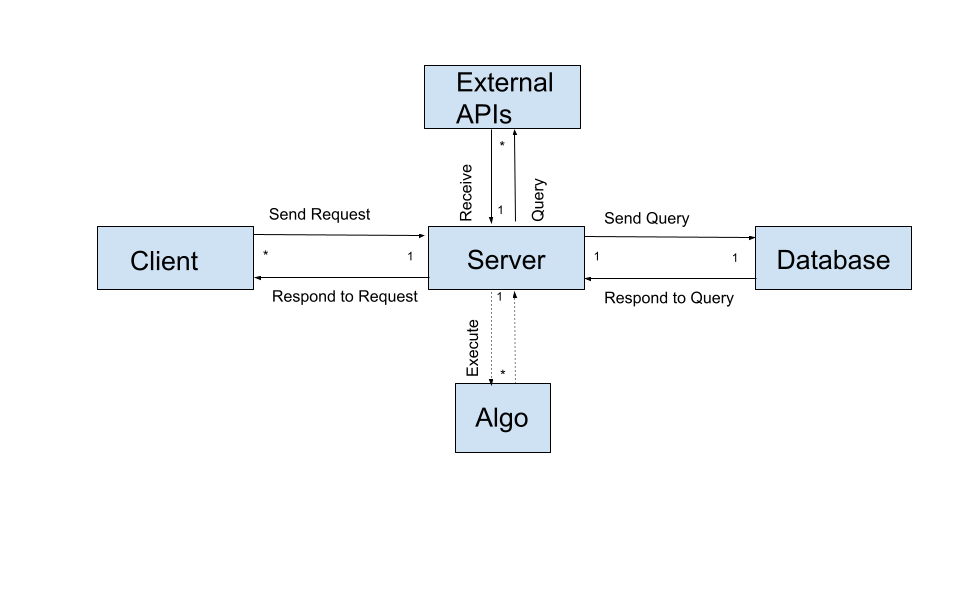
The interface should be clear to understand and simple. Having the options to navigate the website while also minimizing the amount of pages required to use the website. When creating a schedule, users should have clear indications of the selections they have made before and after constructing the calendar so that it can be easily referred to. Being able to refer back to one’s schedule should also be a possibility through saving the schedule and associating it with the user’s login will make ease of access and avoid having to recreate the schedule each time.

1. Responsiveness for use on different devices and screen sizes
2. Adherence to web accessibility standards
3. Average interface response times of non-algorithmic dependent functionality should be around 200 milliseconds and not exceeding 1 second

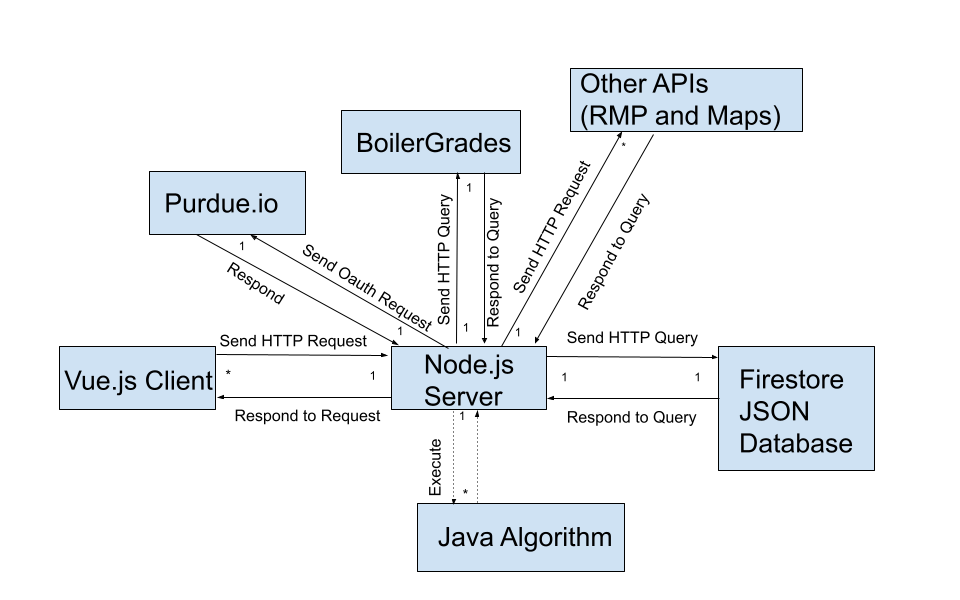
# 

# Design Outline

Our project follows a client-server model to provide optimal class schedules to Purdue students. The server will have access to a database that contains cached user data such as their previously designed schedules, class interests, and statistics on professors and class availability.



1. Web client
   1. The web client written in Vue.js is the sole place for the end-user to interact with the BoilerTime service. When users visit the website, they will have the ability to interact with the front-end UI which has the ability to make queries and render data as appropriate.
2. API server
   1. The API server written in Node.js acts as an intermediary between the services that BoilerTime operates with. When receiving a request, the server parses the request and validates the data before storing it into the database or passing it into the algorithm with a schedule request.
3. Database
   1. Our database, Firestore, will store data that isn’t updated regularly, for example average professor GPAs from BoilerGrades. We will also store classes from Purdue.io that are offered in the current semester.
   2. The database will respond to requests for information from the front end via the API server to render pages that require profile information, and will respond to the scheduling algorithm via the API server during the process of generating a user schedule.
4. Scheduling algorithm
   1. The algorithm, written in Java, is invoked by the web server upon the web server receiving a request from a client.
5. External APIs
   1. We will make use of external APIs including Purdue.io, BoilerGrades, RateMyProfessor, and Google Maps to handle schedule optimization. These requests will be staged by the API server and forwarded to the algorithm once all appropriate data has been fetched



Exploring the high-level relationship, there are many clients (created using Vue.js) who communicate with a single Node.js server which acts as the staging point between the various services offered. There is a 1:1 relationship between the server and the database because only one server is required, and only one database would make sense so as to maintain data coherency.

# Design Issues

## Non-Functional Design Issues

### Which framework/language for frontend?

1. Vue.js
2. React.js
3. Django

**Choice: Vue.js  
Justification**: Vue.js has a wide range of tools and offers a great amount of flexibility for designing websites. Alongside this, it reportedly ranks as the third most used JavaScript framework as of 2023. This helps us as it is very well documented and easy to learn from online sources. Vue.js allows for many unnecessary steps to be removed from the process and will allow for greater efficiency in the development process, specifically Vue’s lifecycle hooks and object rendering makes it easier to present the JSON data returned by our API using HTML. On top of this, Vue.js functions extremely quickly, and is noted to be one of the fastest web development frameworks. Lastly, the Virtual DOM that Vue uses integrates a diffing algorithm that allows for the framework to be more lightweight and results in a significant speed and performance increase. We ultimately chose Vue.js because of the HTML-centric compatibility and the high speeds it can offer.

### Which UI Library?

1. Tailwind CSS
2. Bootstrap
3. Material UI

**Choice: Tailwind CSS**

**Justification:** Compared to other UI libraries, CSS frameworks specifically, Tailwind offers much more control when it comes to styling the website, which means there is no required or default theme. This not only allows for the website to have a more pertinent identity, but also allows for developers to make decisions that best fit the objectives of the website while not being shoehorned into a specific layout. As Tailwind utilizes pre-built classes, you can design the website’s layout in the HTML file, which makes it more responsive and works well with mobile devices. In comparison to Bootstrap, Tailwind allows for a greater amount of customization, it is more modular due to being implemented as utility classes in HTML, and has a lighter file size.

### Which language for backend?

1. Java
2. Node.js
3. PHP

**Choice: Node.js**

**Justification:** Node.js is known for being the fastest out of these three, especially when it comes to quick response times. Our website will have lots of requests and we will need to access the database and user input often. This is the most efficient via Node.js. Node.js is also designed for scalability; many connections at one time can be created on Node efficiently. Node is also a language that has lots of packages especially for web development. This will help us find libraries and use already well written code. A lot of us already have experience with JavaScript as well so the learning curve will not be very high.

### Which language for the algorithm?

1. Java
2. C++
3. Python

**Choice: Java**

**Justification:** While C++ is the fastest out of the three, we are not very familiar with C++ external libraries and Object-Oriented Programming. Because we lack familiarity with the language, the ACM Code of Ethics tells us not to use tools we are unfamiliar with. Python is not viable because its performance when using NumPy and Pandas to complete calculation intensive tasks is not scalable for the nature of the project. Java offers a compromise of being comparable to C++ in both features and execution speed while still being familiar to us through our coursework at Purdue and outside institutions.

### Which database?

1. Firebase Realtime Database
2. Firestore
3. MySQL

**Choice: Firestore**

**Justification:** We chose Firestore because of its ability to provide scalability in an easily used and maintained format. First, we eliminated MySQL because of its use of a structured query language, which does not conform with our requirement of easily supporting hierarchical data – that is each class, TA, or student record can have child nodes containing more specific information about the courses, ratings, or preferences. Because the number of children is different for every child – and, therefore, not known at the time of database creation, using SQL is not a reasonable option. Firebase Realtime and Firestore are both viable options as they are NoSQL-based models, which supports the type of data that is used by BoilerTime. Firestore is preferable over Firebase Realtime because it offers more easily segmentable data and faster searching time through archives of data that must be read frequently when generating schedules but rarely receive updates (ratings, for example, need to be read on the creation of every new schedule but only need to be written to occasionally).

## Functional Design Decisions

### Which data point from BoilerGrades?

1. Average GPA
2. Percent Fail/Pass
3. Percentage of A/A+

**Choice: Average GPA**

**Justification:** Average GPA provides a strong middle ground that allows the algorithm to include the interests of all students without requiring a large amount of data to be stored. Because of the large volume of courses and professors at Purdue, storing multiple datasets would be costly both in terms of storage resources but also in searching time when data is being fetched for use by the optimizing algorithm. We reject pass/fail because it provides too little insight into the overall grade breakdown when people are making decisions – students would like the option of receiving a score that is above just passing. We reject the use of percentage of A/A+ in the course because receiving a grade of A/A+ is will only occur for a small subset of students. The use of average GPA helps to create a more representative system for all BoilerTime users because it covers more of the grade distribution, and, as a result, more of the user base who typically lie across the grade distribution.

### Which format of displaying schedules?

1. Calendar
2. List
3. Map

**Choice: All**

**Justification:** As this would typically be a user preference, we decided to offer all the options to meet user accessibility and convenience. In doing so, we provide a more comprehensive and versatile user-facing experience to ensure that the data can most effectively be used by any given individual. After all, the data being received from the back end is composable and reusable in nature. This means that it would not take up any significant additional resources to produce the three different formats.

### Login Information?

1. Hashed Passwords
2. Outlook
3. No Login

**Choice: Hashed Passwords**

**Justification:** We chose hashing passwords because it balances security with the ability to have user accounts. First, we eliminated not having a login option because it would not allow users to share data across their devices, would not allow them to create shareable links with their friends, and would not require any authentication to leave reviews on TAs and classrooms. Next, we eliminated Outlook because it is not user friendly for a service, especially for a service that does not have sensitive personal information that could benefit from two factor authentication. Hashed passwords provide the best solution because they are simple for the user to create and account and login with, and thanks to hashing are secure against malicious access on the server. Because Purdue email addresses are collected, it is possible to verify that all users are legitimately members of the Purdue community before they may leave reviews and share other information with the site.

### Display detailed information about a course and instructional staff?

Summary: This is a question of whether/how to display information such as detailed graphs and reviews of courses to end users on the schedule generation workflow.

1. Not at all – users can view details elsewhere
2. On course schedule overview page
3. By clicking on the schedule entry
4. With a modal accessible through the schedule entry

**Choice: Through a modal accessible through the schedule entry**

**Justification:** Users will want the ability to see detailed breakdowns of grade distributions, a class description, and specific feedback on the lecture professor and supplementary TAs when making a course decision. Hence, we eliminate option one because. Next, we eliminate option two because it is too direct – that is the schedule overview (calendar view) should be simple and present only information related directly to the times and locations. Including specifics about the class instructional team makes the UI too confusing and requires too many calls to the database just to render what is essentially a home page. Clicking on the schedule entry similarly should only provide a basic overview of the course – the name of the professor, seats available, and cursory statistics on the professor such as the average number of ratings to maintain simplicity of the UI and to prevent over-use of the database when the user does not actually need to be presented with a large amount of data. Option four provides the best balance because it allows the user to stay within the BoilerTime service while not presenting the details too early to the point of creating an overly complex UI that is computationally expensive to maintain.

# Design Details

## Class Diagram

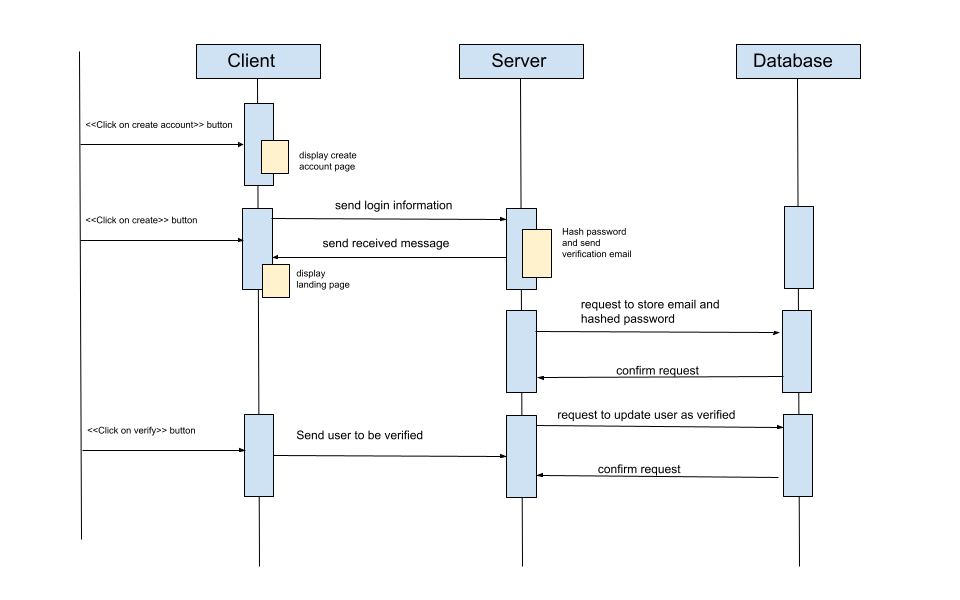
## 

All classes follow a JSON model for data storage, allowing us to maintain a direct relationship between data stored in the NoSQL database and the data that is operated on by the server. The UniqueID service will use SHA to generate unique hash-based IDs that will be stored as strings to make for easier searching and relationships – that is, searching by an ID is easier than by name. The top-level data and classes are TAs, Courses, UserProfiles, and Professors. There can be many instances of the corresponding data – one for each of the many unique individuals in the population that exist, which can be modified using the appropriate getter and setter methods listed. Second level information, TA ratings, lectures, UserSchedules, and ratings, respectively, form a many-to-one correspondence with their first level counterpart because only one first level entry (TA, course, etc…) may have many ratings or lectures. By calling the appropriate creator method in the corresponding top-level entry, a new second-level dataset can be inserted.

## 

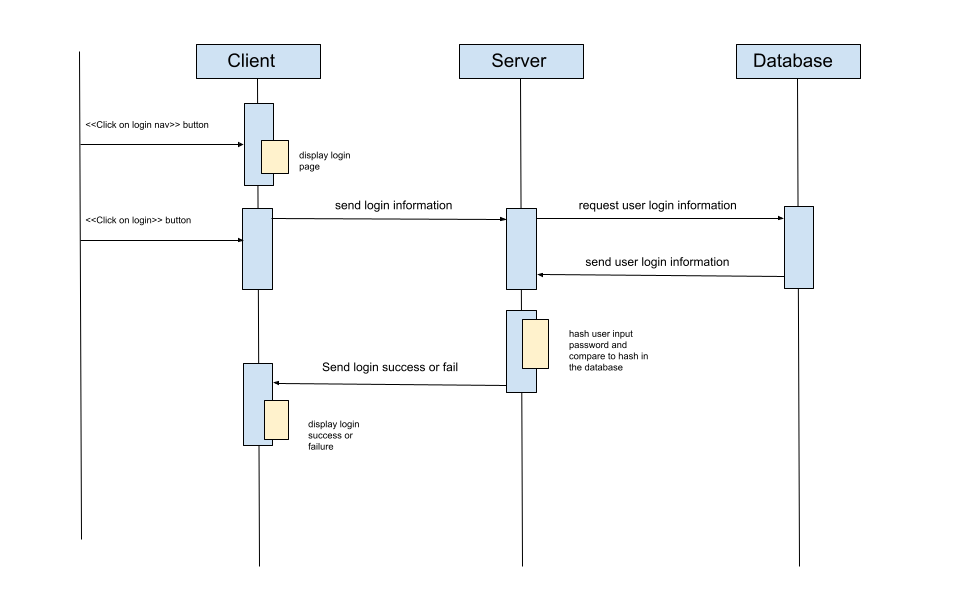
## Service Execution Diagrams

Create Account Service Flow:

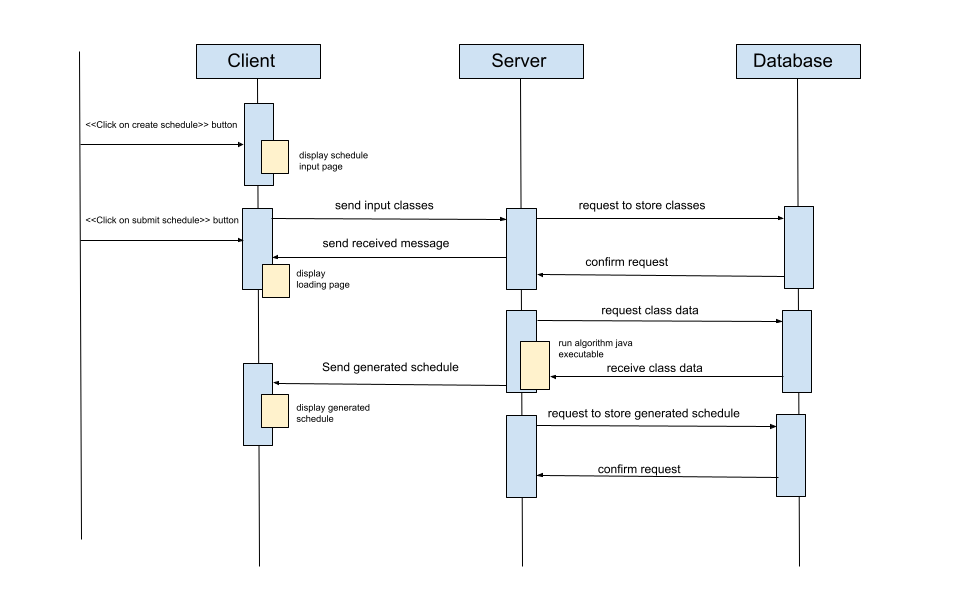


After the user clicks on “create account”, we display a page where the user can input their email and password. Then if they click on the create button, the information is sent to the server which will hash the password and send an verification email to the user. The database will then store the email and hashed password. After the user opens the verification page sent to their email, the server will request the database to update their account as verified.

Login Service Flow:



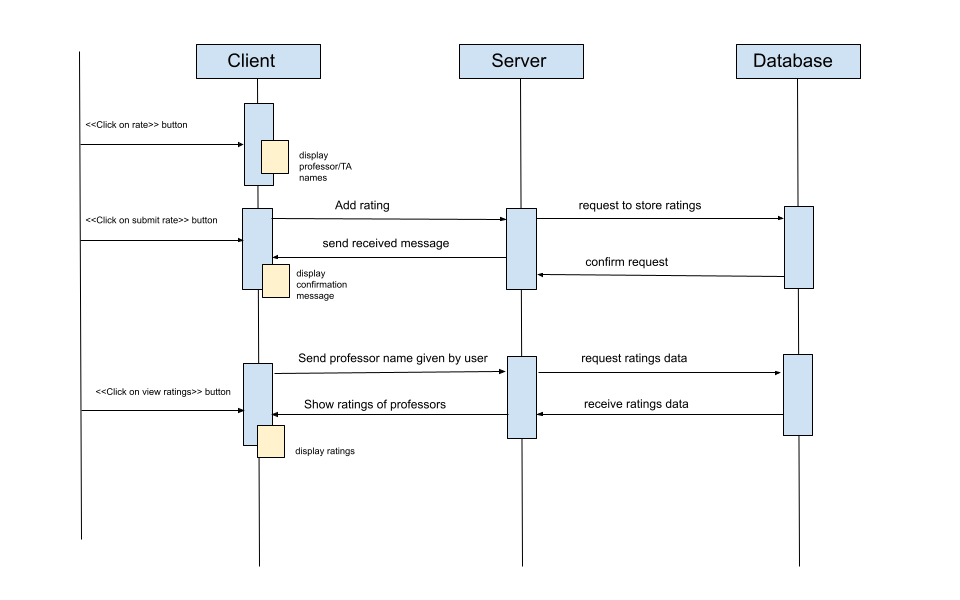
When the user clicks on the login button they are redirected to the login page. Then they will input their email and password. The server will then request the database for the information of the user. If the user does not exist the server will send a failure to the client. If the user exists then the server will match the hashes of the password and if the login was a success or not to the client.

Schedule Service Flow: 

## When the user clicks on the create schedule button, the user is redirected to a page where they can input their required and optional classes. After the user clicks submit schedule, the list of classes are sent to the server which requests the database to store them. Then the server will request the data for the classes submitted by the user to use. The server will run an executable that generates the optimal schedule. This schedule is then stored in the database and sent to the client to be displayed in the calendar.

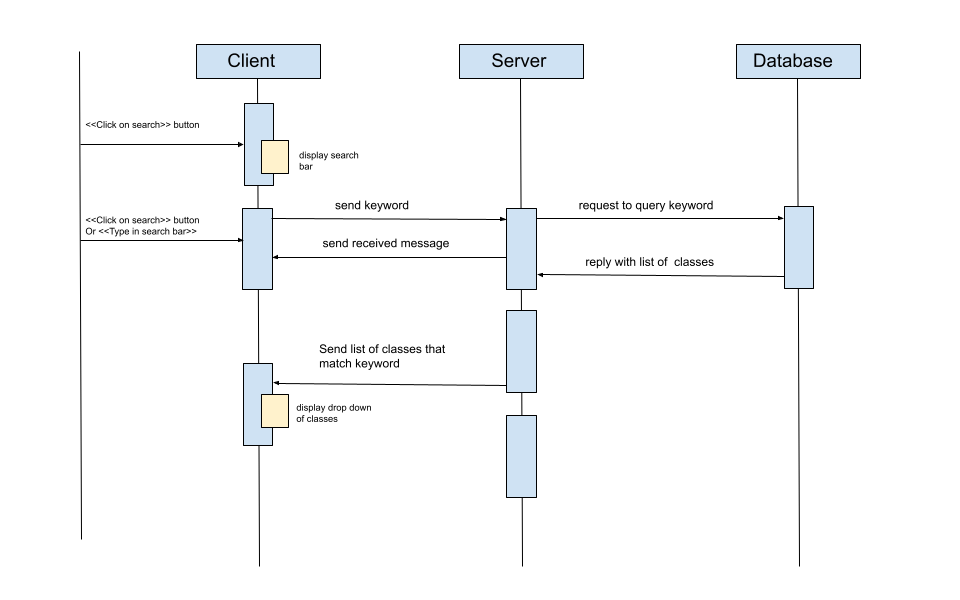
## 

## Ratings Service Flow:



The user clicks on the rate button which will display professor and TA names. After the user fills in the rating and comments, these details will be sent to the server. The server will then store the ratings in the database. If the user wants to view ratings of a certain person. The client will send the server the name of the professor. Then the server will query the database for the ratings which will be sent back to the client through the server.

Search Service Flow:



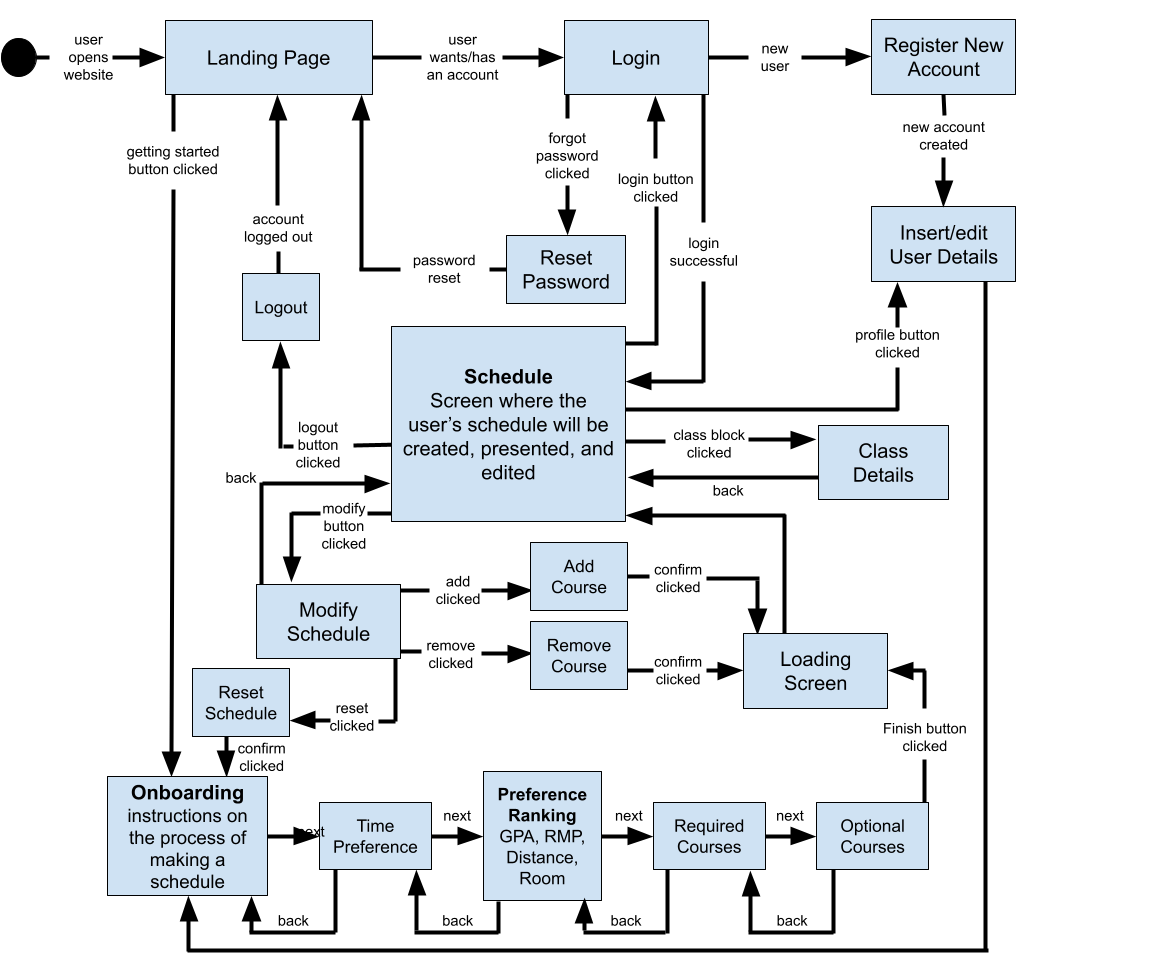
When the user clicks on the search bar a search bar will appear. As the user types in the search bar the client will send the keywords to the server. Then the server will query the database with the keyword and the database will reply with a list of classes that fit the keyword. Then the server will send the list to the client which will display them in a drop down.

## 

## Activity Diagrams

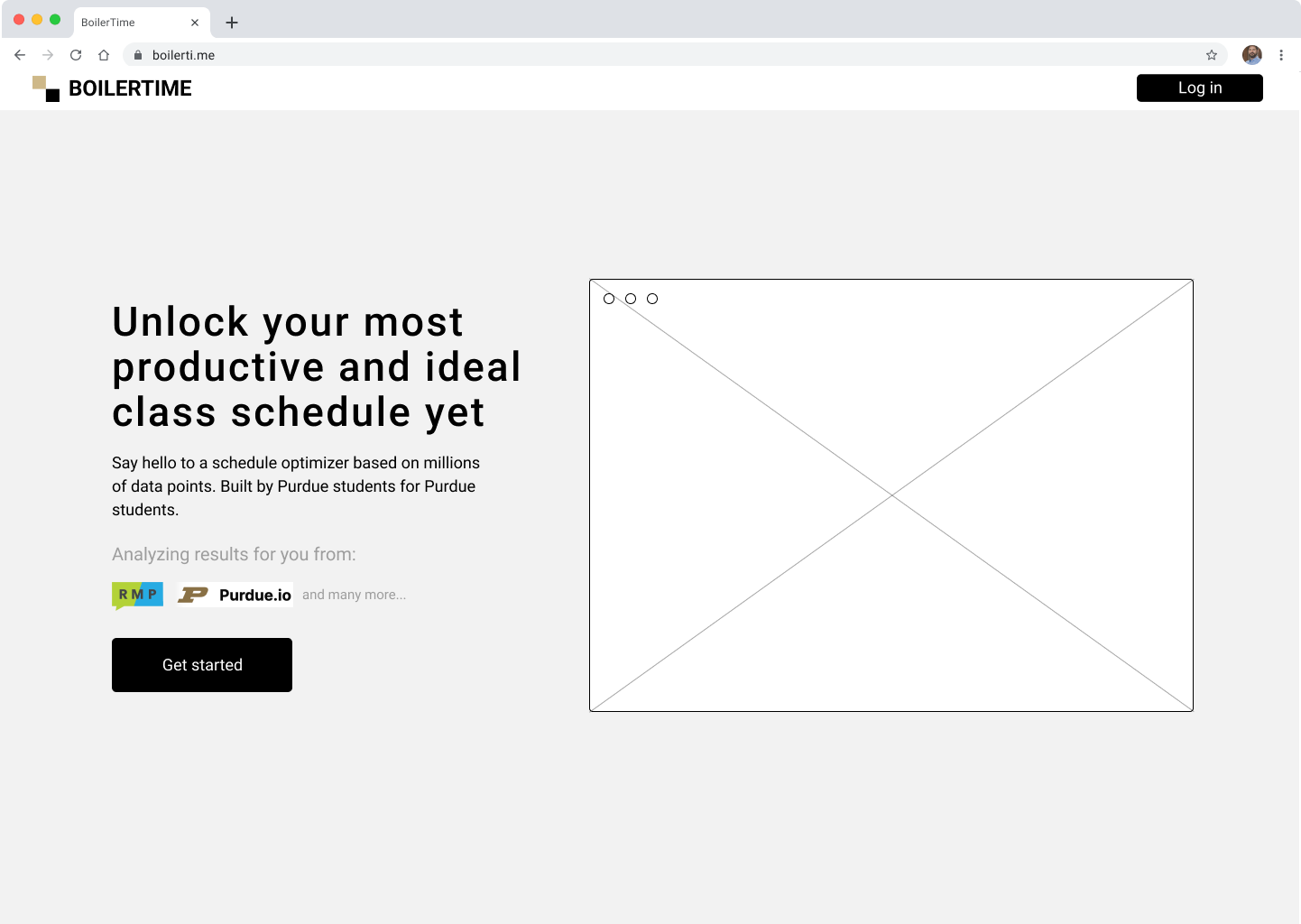
**Navigation Flow Map**

The path the user takes while using the website is focused around a linear track of steps in order to streamline the process and minimize the amount of confusion that could arise, but at the same time, it is important to let users go back to edit selections to help users save time. At the landing page, users have the option to select between logging in and getting started. Getting started is intended for guest accounts, whereas selecting login will allow users to make a new account or login to a pre-existing account. The rest of the diagram is broken into two main components: onboarding and the schedule. Onboarding is the setup for creating a schedule, where users will state their preferences, required courses, and optional courses. The schedule section is where users can view their schedule, edit it, and look at class details.

****

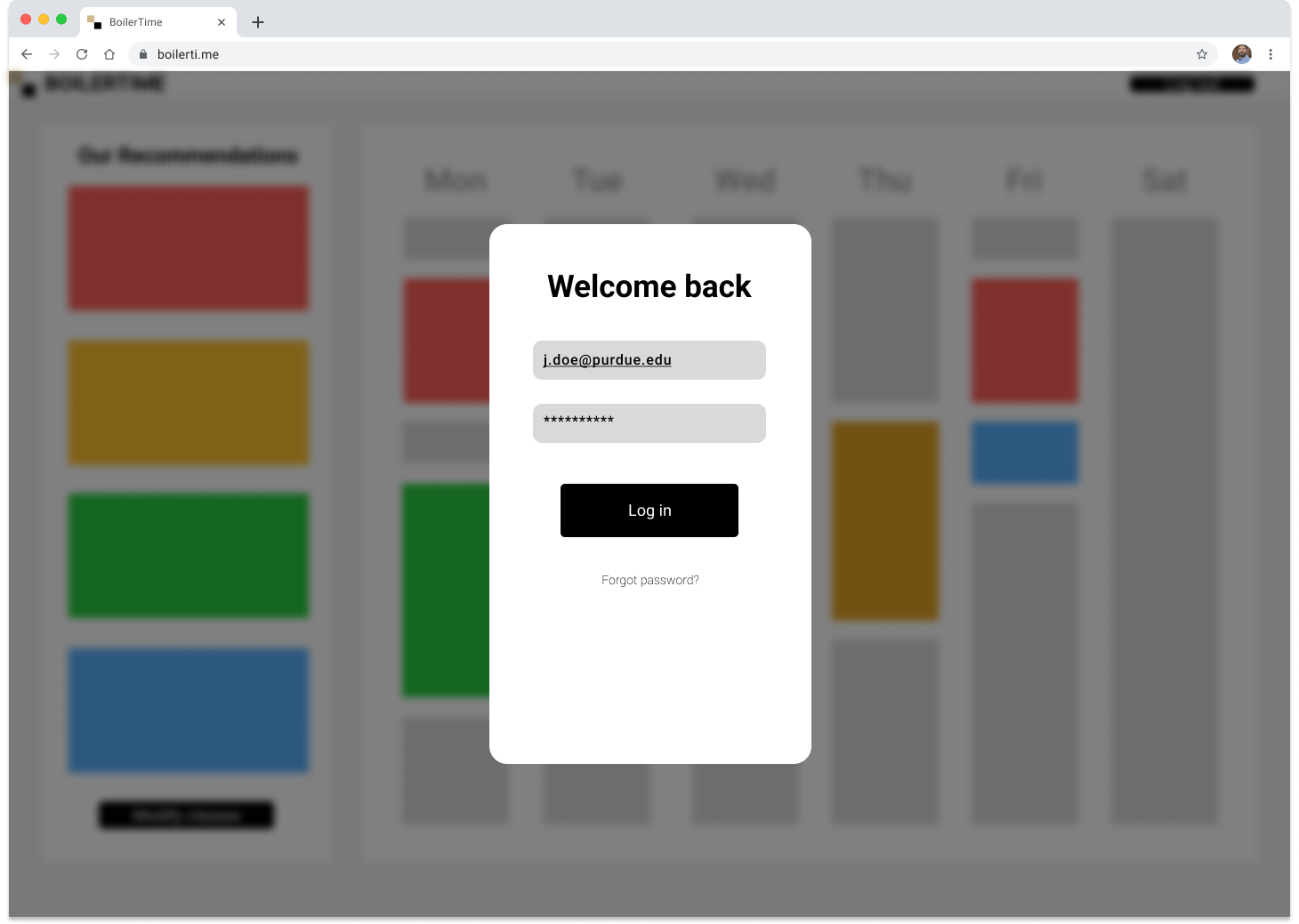
## UI Diagrams

For our design, we have adopted a modern and minimalistic look to the website. We believe that this places full emphasis on the functionality of the website while keeping things simple for the user to understand. Mockups were done using the Figma application. Listed below are samples of the mockups with a short description.



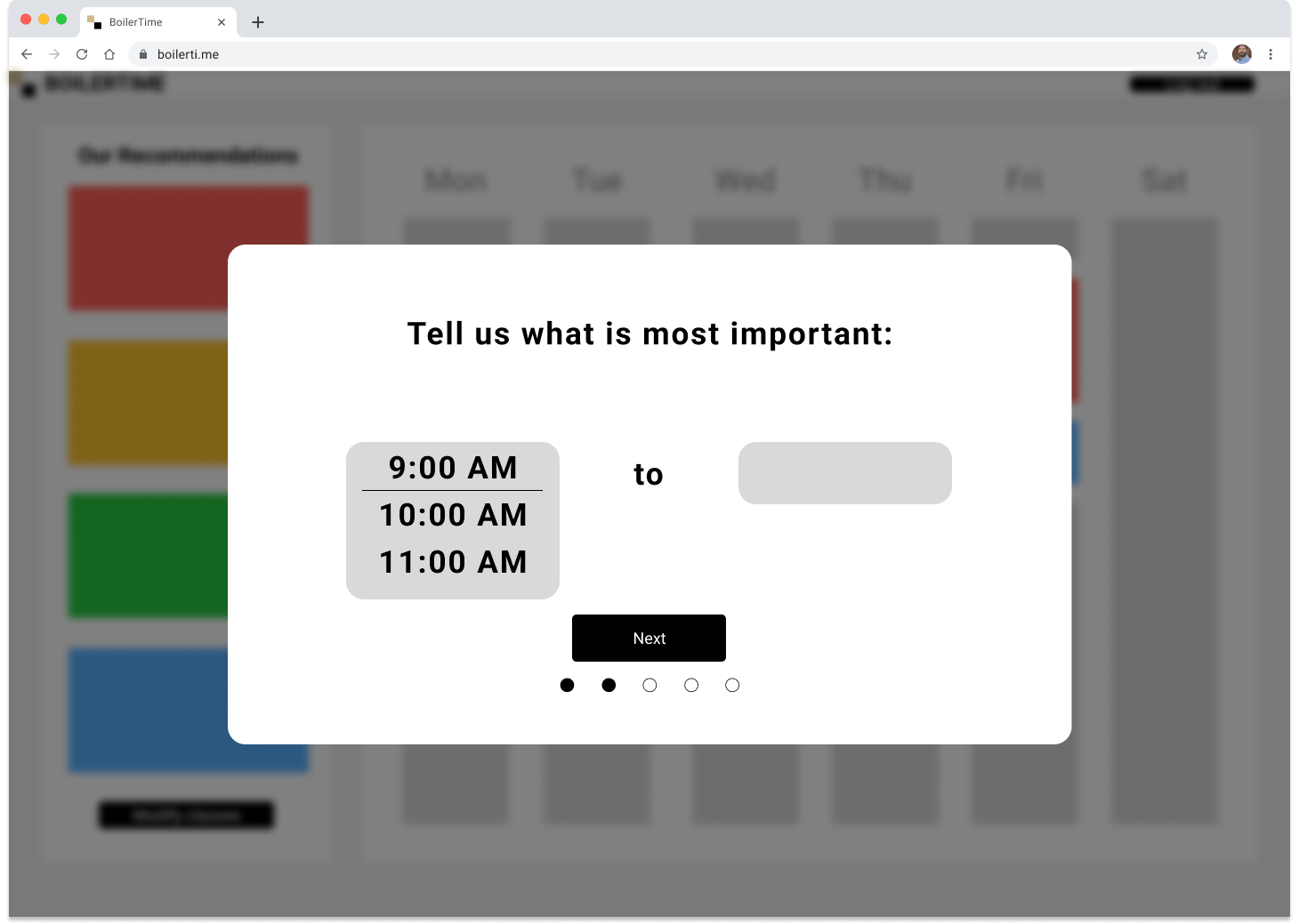
*Figure 1: A photo showing the intro page that the user first views.*

This is the landing page. Representative of the first page any prospective user or visitor of the site will immediately see, it presents the value proposition as well as a call to action on how to begin using the website.



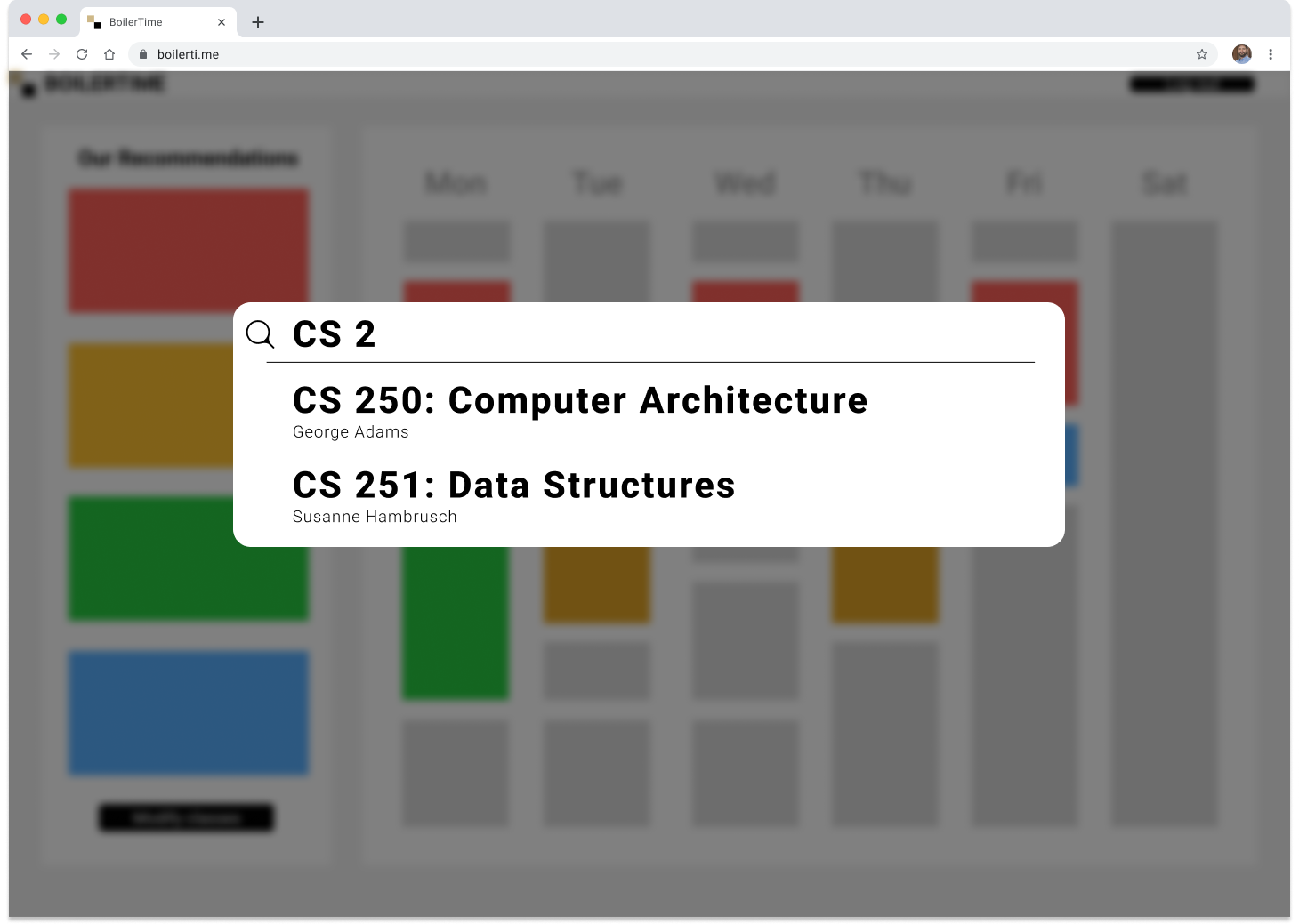
*Figure 2: A photo showing the login window for the website.*

This is the login page. There are text boxes for the user to input their email and password. Once they fill that out, they can click the “Log in” button to log in. There is a “Forgot password?” button included as well in case the user forgot their password.



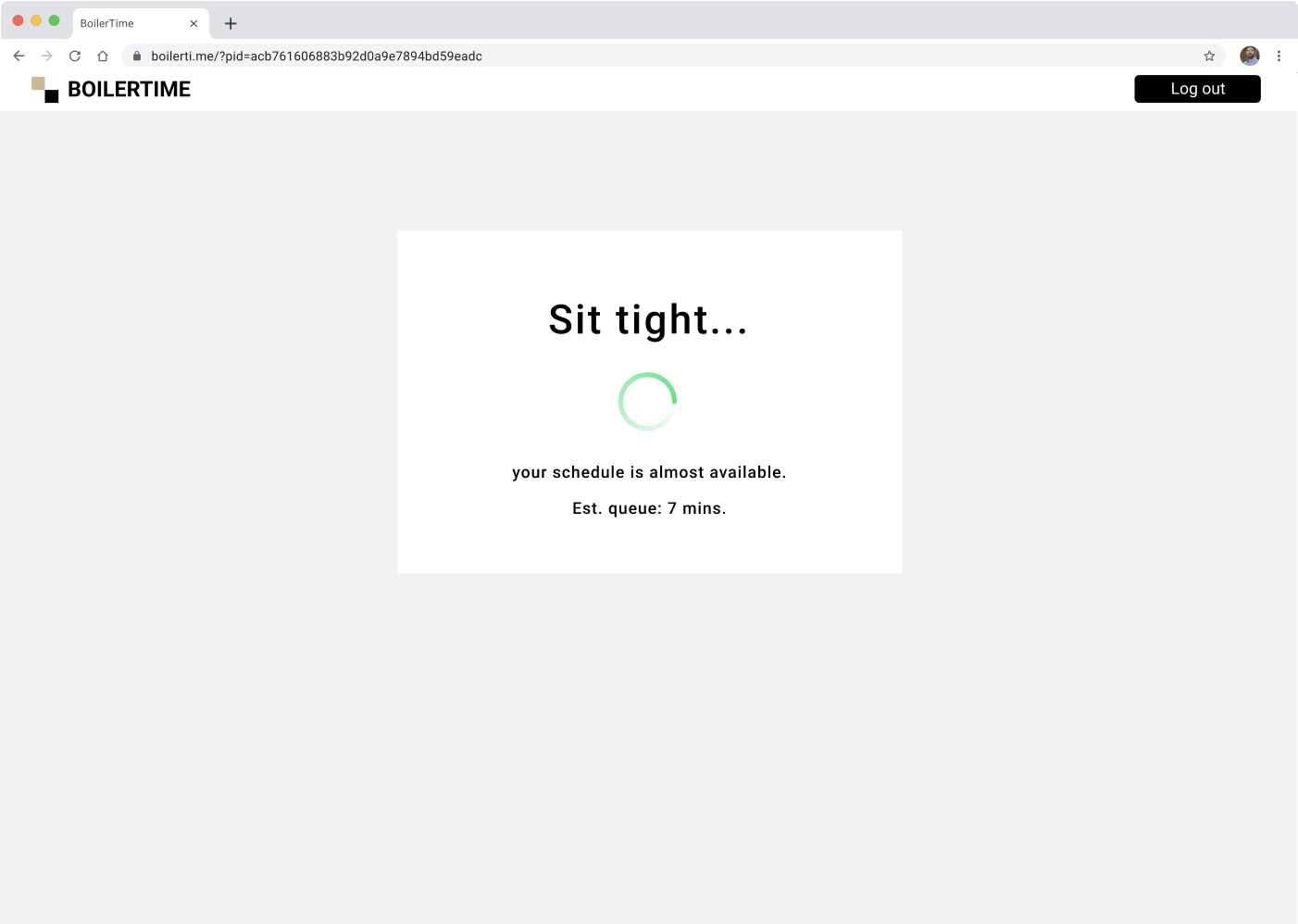
*Figure 3: A photo showing one of the onboarding steps for the initial setup process.*

This is one page of the five pages for the onboarding section of the setup process. In the onboarding window, the user will be led through five steps: instructions for use, time preference, class characteristic importance, required classes selection, optional classes selection, and user profile options. For this page, there are two dropdown lists where the user can select the range for their classes.



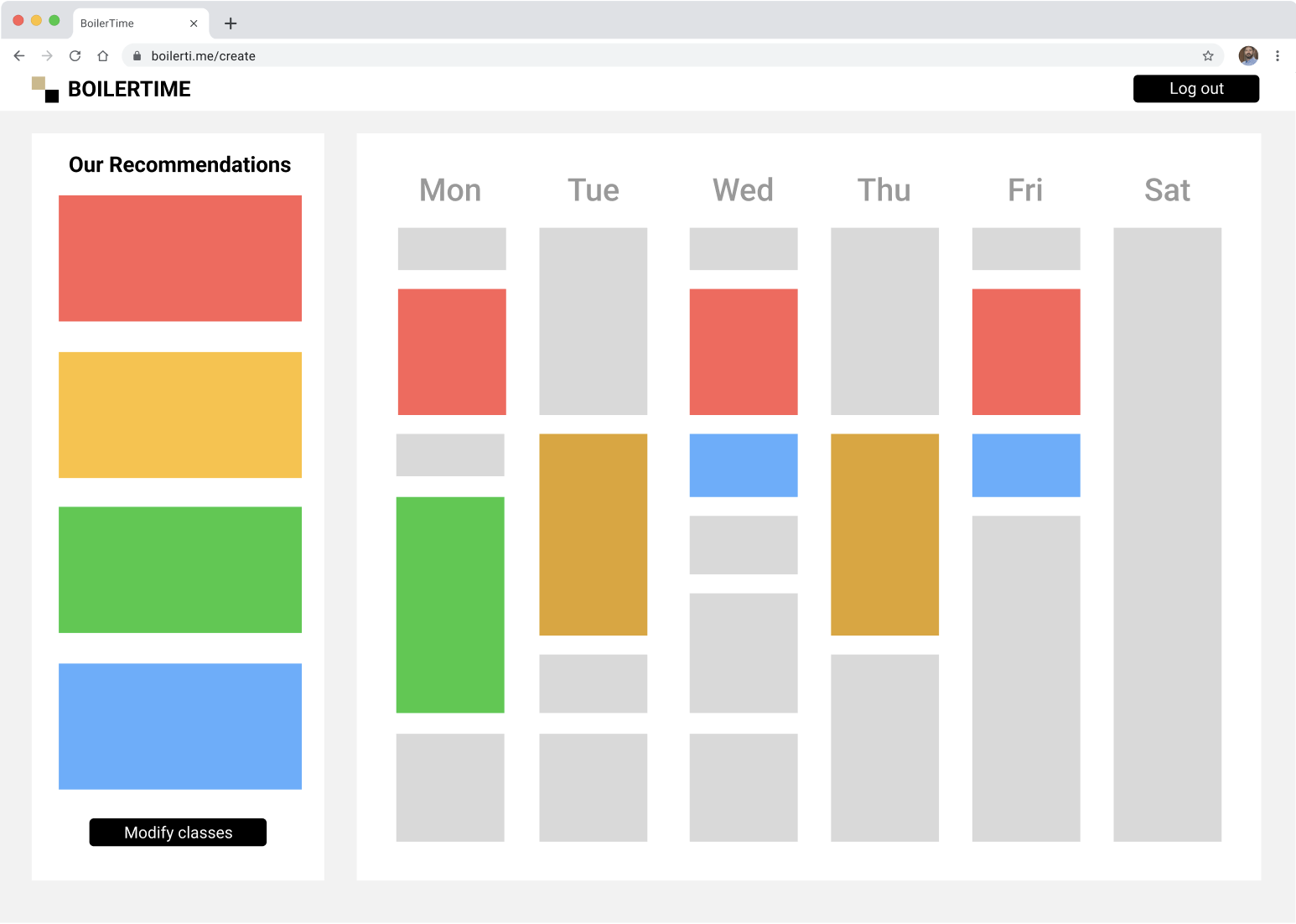
*Figure 4: A photo showing the class search functionality of the website.*

This is the search bar for the website. This search bar allows for the user to type in a class and see the available matching classes based off of the characters they have typed. The user can then select the right class from the drop down list.



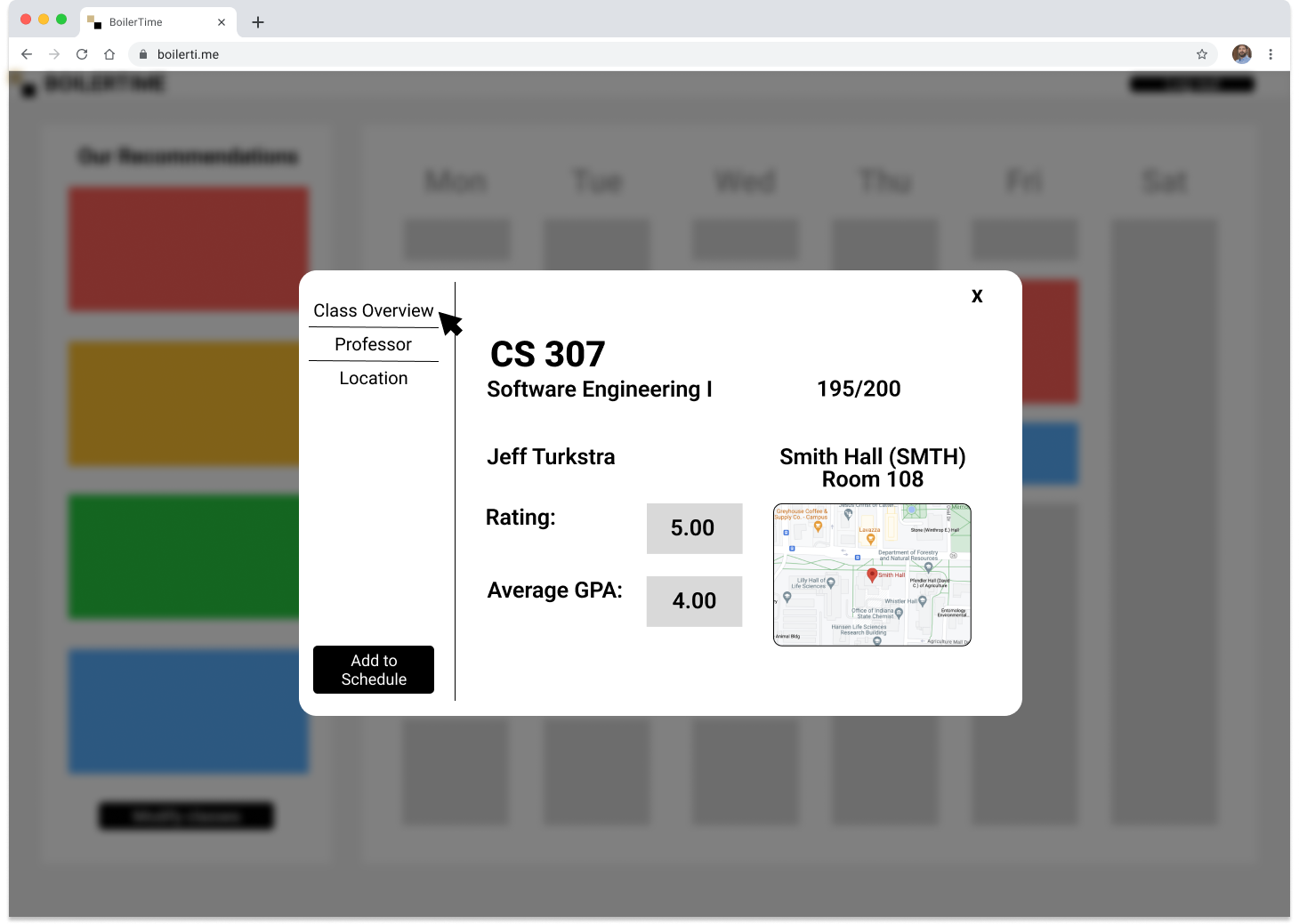
*Figure 5: A photo showing the loading menu while the user’s schedule is processing.*

After a user selects their classes and personal preferences, it will be ran through our back end algorithm. Depending on the number of users subsequently on the website, as well as the number of preferences input by the user, they will be presented with a loading screen.



*Figure 6: A photo showing the schedule that the user views after processing.*

Once our back end algorithm completes factoring in all of the user requests, a weekly schedule will be presented to them. In this screen, the user will be able to view the optimal times for each of their class in a list view and a calendar view.



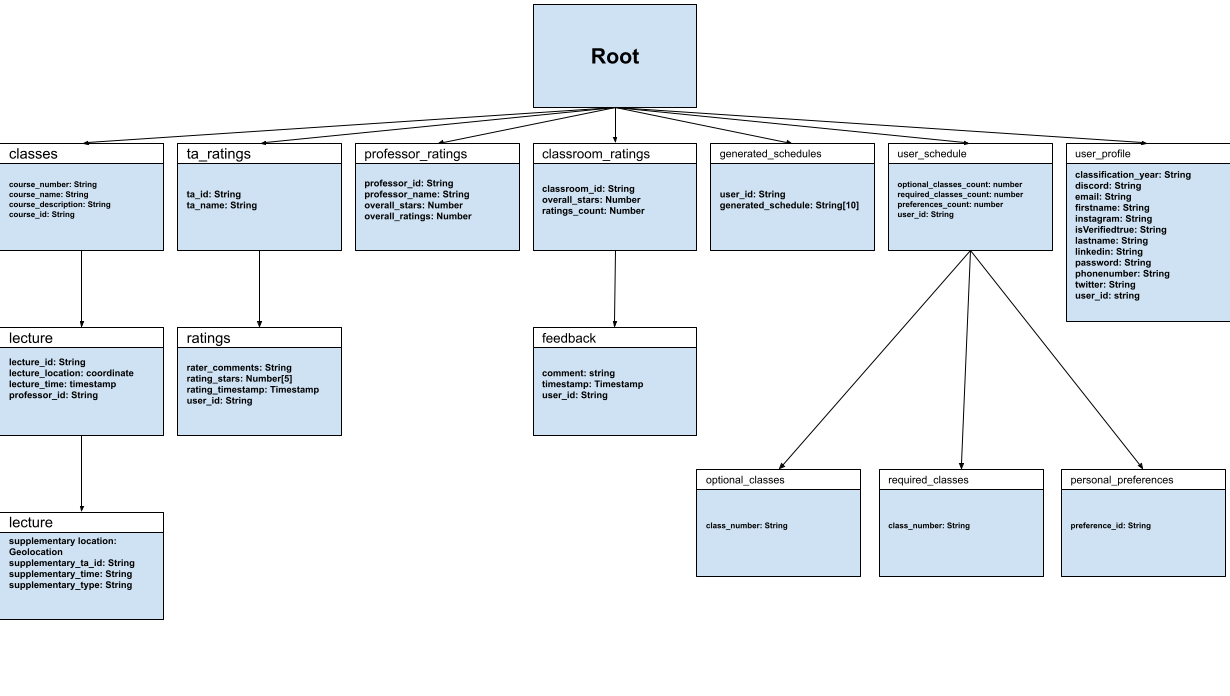
*Figure 7: A photo showing the details of a class that the user clicks on.*

This is the class details box. Shown is the corresponding information for the class that the user clicks on. There is a button present at the bottom that allows the user to add the class to their schedule if preferred. This button is not present if viewing a class already in their schedule.

## 

## Database Diagram

As previously discussed, this program will make use of a Firestore Database, which is a NoSQL database offered by Google Firebase. NoSQL follows a JSON data model, allowing for each node to have children that contain a greater level of detail about a parent node, even when the number of children is not known at creation time. For example, a new “feedback” node may be created every time a new TA review is added, or a new “user\_profile” node may be created every time a new user registers for the service.



## HTTP Requests

| Request | Method | Fields | Returns |
| --- | --- | --- | --- |
| user/profile | GET | auth\_token: String | The authenticated user’s profile, or error message if the auth token is invalid |
| schedule/current | GET | auth\_token: String | The authenticated user’s schedule, or error message if no schedule is found |
| ratings/view | GET | auth\_token: String  review\_type: enum {TA, Classroom}  optional:  unique\_id: String | Returns a list of ratings if found, or error message if no such ratings exist |
| user/login | POST | user\_name: String  password: String  browser\_id: String | The auth token for the user, or an error message if the login failed |
| user/create | POST | user\_name: String  password: String  browser\_id: String | The auth token associated with the new user account |
| user/verify | POST | auth\_token: string | The successfully verified message or error if the email account could not be verified. |
| schedule/new | POST | auth\_token: String | The new, blank schedule that can be modified |
| ratings/rate | POST | auth\_token: String  rating\_type: enum {TA, Classroom}  entity\_id: string  rating\_body: struct {rating\_fields} | The status of the new review: either posted successfully or failed |
| user/edit | PUT | auth\_token: String  update\_body: struct {modified components} | The status of the update: the changes made if successful or error code if failed |
| schedule/update | PUT | auth\_token: String  update\_body: struct {modified components} | The new schedule or error code if failed to update |