

Seed and Lead Impact App Design

JIC Team 1104 (H-Brans):

Steven Chen – stevenchen@gatech.edu

Anh Tuan Ho – aho71@gatech.edu

Brent Murphy – bmurphy49@gatech.edu

Harrison Smith – hsmith323@gatech.edu

Hannah Ward – hward9@gatech.edu

Client:

Nicole Ford – fordn@fultonschools.org

GitHub:

<https://github.com/harrisonrsmth/SLI-App-H-Brans-1104>

Table of Contents

<i>Table of Contents.....</i>	<i>2</i>
<i>Terminology</i>	<i>3</i>
<i>1 Introduction</i>	<i>5</i>
<i>2 System Architecture</i>	<i>6</i>
<i>2.1 Static Elements</i>	<i>6</i>
2.1.1 User Interface.....	7
2.1.2 RESTful API	7
2.1.3 Business Logic.....	7
2.1.4 External Libraries.....	8
2.1.5 MySQL Database	9
<i>2.2 Dynamic Elements</i>	<i>9</i>
<i>3 Data Storage Design</i>	<i>11</i>
<i>3.1 Database Use.....</i>	<i>11</i>
3.1.1 Database Schema	11
<i>3.2 File Use.....</i>	<i>13</i>
<i>3.3 Data Exchange</i>	<i>13</i>
<i>3.4 Security</i>	<i>13</i>
<i>4 Component Design</i>	<i>14</i>
<i>4.1 Static Design.....</i>	<i>14</i>
<i>4.2 Dynamic Design</i>	<i>16</i>
<i>5 UI Design</i>	<i>17</i>
<i>6 Appendix A: RESTful API Documentation.....</i>	<i>24</i>
<i>6.1 Response Codes</i>	<i>24</i>

Terminology

AWS RDS:

Amazon Web Service's Relational Database Service which hosts MySQL databases.

Backend:

The portion of the web application that the user does not see where the logic and data storing takes place such as the background processes that take place when a user clicks a button.

Bootstrap:

A frontend framework that provides CSS and JavaScript based templates for interface components.

Carbon footprint:

The total amount of greenhouse gases that are generated by our actions.

Carbon handprint:

The positive environmental impact of a product or service throughout its life cycle.

Entity Relationship Diagram (ERD):

A UML diagram used to illustrate entities, attributes, and relationships among entities of a relational database.

Flask:

A web framework, a Python module that lets you develop web applications easily.

Frontend:

The portion of the web application that the user interacts with such as the screens with buttons and text boxes.

Heroku

Service where we deploy our front and backend code to be executed.

JavaScript:

A high-level programming language that is often used in frontend web development

JSON:

JavaScript Object Notation – a lightweight format for storing and transporting data.

Node.js:

JavaScript framework used to build scalable network applications.

Python:

An interpreted high-level coding language which emphasizes readability and helps programmers write clear and logical code.

React:

A JavaScript library for building user interfaces.

Sustainable Development Goals (SDGs):

17 goals defined by the United Nations to better and protect our environment around the globe.

SQL:

Structured Query Language, used for extracting and organizing data in relational databases.

1 Introduction

The purpose of this project is to provide kids, grades K-5, with a way to log environmental service that they have done for the community. Our client, Nicole Ford, works closely with her students, encouraging them to better the environment around them. With the Seed and Lead Impact Application (S.L.I App), we aim to allow teachers to create campaigns/assignments for community outreach and students can record their work accordingly.

Our app will provide the user with a space to record custom service campaigns and have the ability to display which SDGs a particular campaign contributes to. With this new application, students will have the opportunity to learn the value of community and environmental service from a much younger age. The idea of being able to visualize their efforts' effects with the user interface is intended to keep the students motivated and spark a genuine sense of pride in their service.

2 System Architecture

2.1 Static Elements

The S.L.I. App will utilize a layered architecture that is composed of four major layers. The architecture is partitioned in such a way to ensure that responsibilities among the components are kept separate in a way that is easier to manage and consistent with the design principles of high cohesion and low coupling. The static system architecture shown in Figure 2.1 displays how each of the components interacts with the others. The arrows represent dependencies, and therefore, expected return values have been abstracted away. The following subsections will describe each component in detail illustrating what is included and how each one operates.

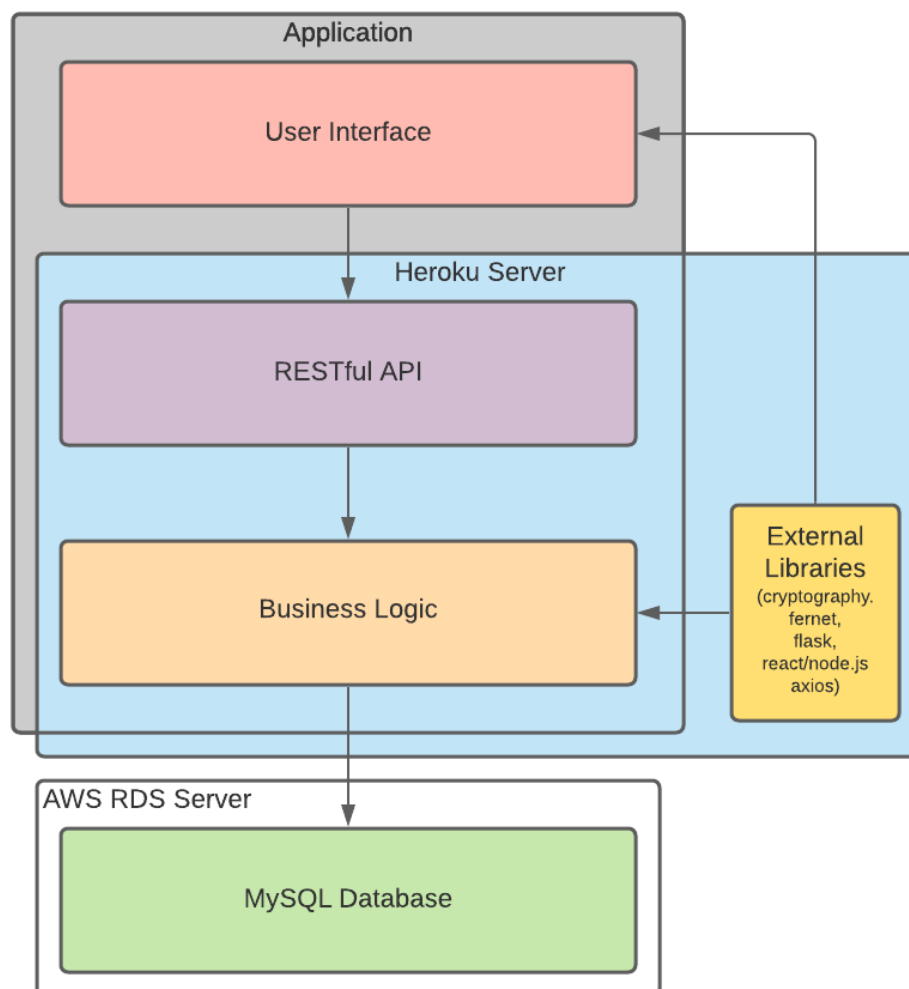


Figure 2.1: Static System Architecture for Seed & Lead Impact Application

2.1.1 User Interface

The user interface, or frontend of the web application, is built using the React framework. The frontend is made up of components, or screens, that the user sees and interacts with. The components are generated using Node.js libraries and Bootstrap HTML and CSS templates, which are then customized to fit our needs. Each of these components is a separate, highly specialized, and independent element of the codebase, demonstrating the principles of low coupling and high cohesion.

The user interface is responsible for collecting user input data so that it can be passed to the backend via the API for processing. For example, the login screen prompts the user to input their login information, which it then collects and passes to the backend in order to authenticate the user. After a call to the backend, the interface will typically be required to redirect to a new screen as well.

The user interface also provides the user with all the necessary information the user would need regarding their activity within the application, such as their login status and which account they have logged into. Depending on whether the user is a student or a teacher, the interface will vary and offer different functionality to the two distinct user groups such as teachers having the ability to create a class but students not having that option.

2.1.2 RESTful API

The RESTful API serves as the intermediate step when passing data from the frontend to the backend to ensure uniformity of data in the frontend and backend. The API defines what is needed from the frontend in order for backend calls to be able to execute and what is expected from the backend once the backend returns to the frontend in order for the frontend to render properly. What makes it RESTful is the fact that information is passed into the API from the frontend client to the backend server and each request is separate and unconnected.

The API uses the JavaScript package `axios` to make GET or POST requests to the backend logic. `Axios` is an HTTP client for `Node.js` and is Promise-based, meaning that the API call expects a return value of whether the request was fulfilled or rejected.

Any input data needed from the frontend is passed through the API in JSON format so that the backend function can know what to expect to be passed as input. Then, upon completion of the backend functions, JSON data is returned back through the API to the frontend, where the frontend expects certain pieces of data to be returned.

2.1.3 Business Logic

The business logic for the application is written in Python, making it simpler to understand from a high-level. It is hosted on a separate Heroku server using Flask to have a Python-specific backend framework. Each function in the backend is routed via a distinct URL path so that when the API makes a request to a specific URL, it calls the intended function in the backend, demonstrating the intended controller design principle.

of our architecture. The backend also utilizes a flask library called `flask_cors` which ensures that none of the backend functions can be called without receiving the proper request from the API.

The backend functions are responsible for taking the input from the frontend that was passed through the API and processing it so that it can interact with the MySQL database. Once the functions have successfully completed, they must construct the JSON data that will be expected by the frontend upon return. Such data typically includes a response code (200 for success), and entries to represent the state of the application such as the user that is logged in.

2.1.4 External Libraries

The application relies on several external libraries, primarily in the backend.

2.1.4.1 *cryptography.fernet*

This library is used to encrypt and decrypt user's passwords within the database. When a user first creates their account, the backend function is responsible for creating accounts encrypts the string password into a byte array with Fernet that is then passed to the database. Then, upon login, when the backend function must retrieve the password from the database to authenticate the login information, the function decrypts the retrieved password again using Fernet. This is done to ensure that no actual passwords are stored in the database and that passwords are kept secure.

2.1.4.2 *flask*

As mentioned before, the backend logic is hosted using the Python Flask framework. Flask is a free open-source library that integrates well with Node.JS. The flask library enables us to create a Flask app that can route functions by URLs that the API can send requests to. Flask also includes several extensions such as `flask_cors` and `MySQL` that enable the Flask app to function properly when working with the other components.

2.1.4.3 *react/node.js*

The react library is used in conjunction with node.js to create the frontend of the application. These two libraries make it possible to construct the components, or screens, of the frontend using Bootstrap HTML along with JavaScript functions to maintain the flow of data through the API. Each component is initialized with a state that can be altered by user input, and then the JavaScript function is able to pass this state data to the API.

2.1.4.4 *axios*

As mentioned before, the API is supported by the `axios` library, which allows the API to make GET and POST requests to the backend logic. It integrates well with Flask and also sets up the retrieval of data from the API once the backend logic returns JSON data back to the frontend through the use of a Promise.

2.1.5 MySQL Database

The main driver of the application is the MySQL database that holds all the information and data necessary for the app to function properly. The database is a relational database that is divided into tables that can reference one another. This makes the data much easier to maintain since it is categorized in a logical and intuitive way. The database is hosted on an AWS RDS server.

The functions that interact with the MySQL database are written in Python and are hosted with the backend functions on a Heroku server. This file makes use of a MySQL Flask extension that makes querying a MySQL database with Python very simple. The functions set up a connection to the database and are then able to directly execute queries to retrieve, insert, or delete data.

When a backend function needs to interact with the database, the backend function simply makes a call to the SQL query function using the connection to the database that has been initialized in the backend file. This provides a very deliberate level of abstraction between the backend logic and the database. It consolidates all instances of the database connection and cursors that execute queries into one location so that it is easier to maintain and trace as the application scales, demonstrating the open-closed design principle—the database is open to expansion but closed for modification.

2.2 Dynamic Elements

The Dynamic System Architecture, seen in Figure 2.2, depicts a scenario where a valid user logs into their account on the S.L.I. App, navigates to the Log Work screen, logs work, then goes back to the Dashboard to view their work logged followed by logging out and closing the application. It is important to note how the low coupling and high cohesion is visible in Figure 2.2 as it is clear that each layer only communicates with its immediate neighbors. First, the user clicks “Login” and enters their login information, which is then verified through our MySQL database, creating a token upon successful validation. Until the user logs off, this token in the MySQL token table ensures their session is active and valid. Upon successful login, the user is taken to the Dashboard screen.

From the Dashboard, users may navigate to the Log Work page to log new work, which will then get stored in the database, or they can view their previous work, which will also be displayed by querying the database for existing data. After new work is logged, the user is then greeted with a confirmation message, letting them know that it has successfully been entered into the database. When the user returns to the Dashboard, they will be able to observe the new work logged in addition to previous assignments. To leave the app, users can click “Logout” and the session will be closed by means of the associated user token being deleted from the MySQL token table. The user will then be taken back to the Login screen of the interface and will be able to close the application.

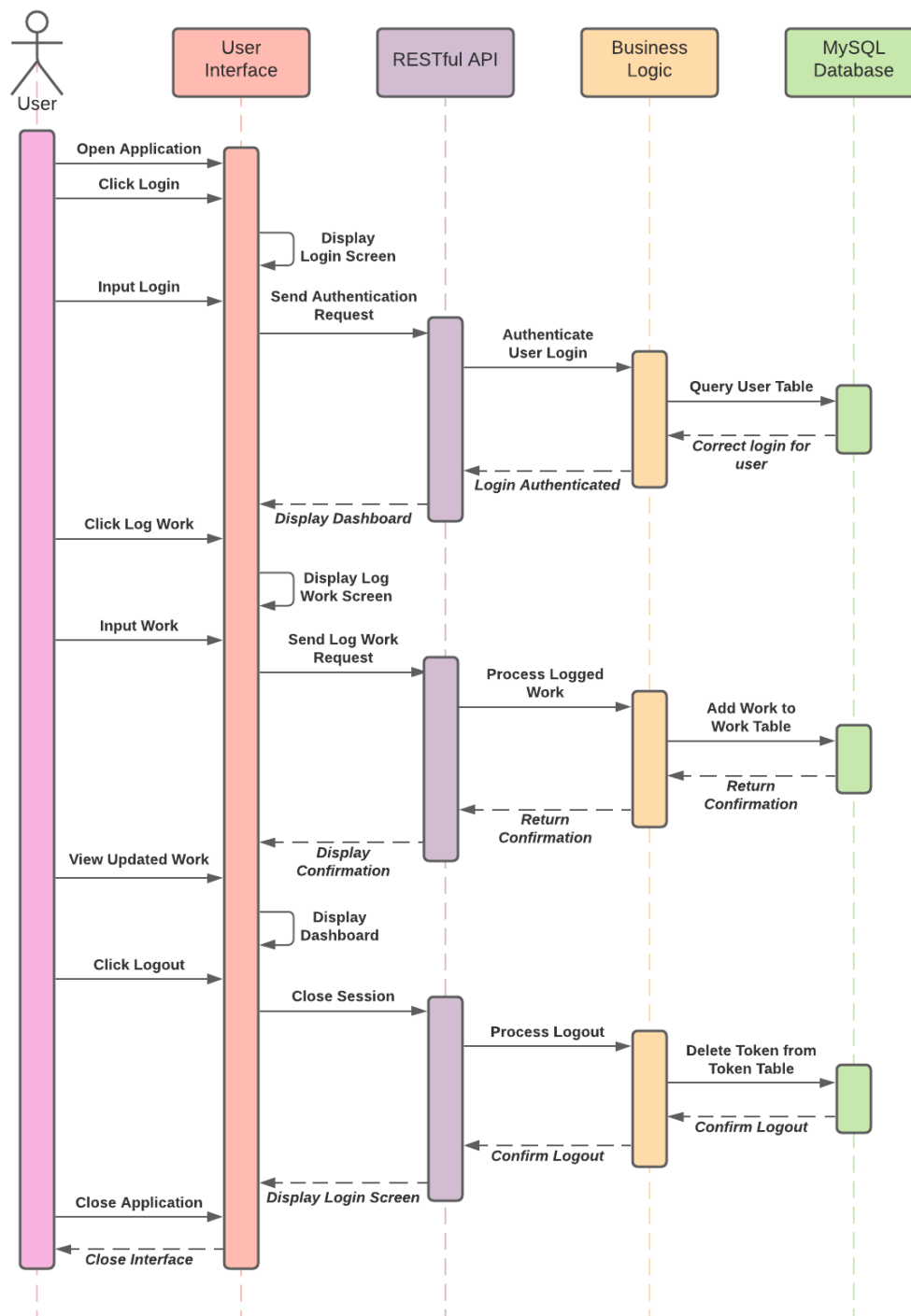


Figure 2.2: Dynamic System Architecture for scenario of logging in, logging work, viewing dashboard, and logging out

3 Data Storage Design

The web application uses a MySQL relational database to store all user and application data necessary for the application to function that is hosted on an AWS RDS server. As seen from section 2.2, the backend logic makes calls to our data layer, which queries the database to either retrieve, insert, update, or delete data. Section 3.1 includes an entity relationship diagram (ERD) followed by a written schema that demonstrates primary keys and foreign key dependencies.

3.1 Database Use

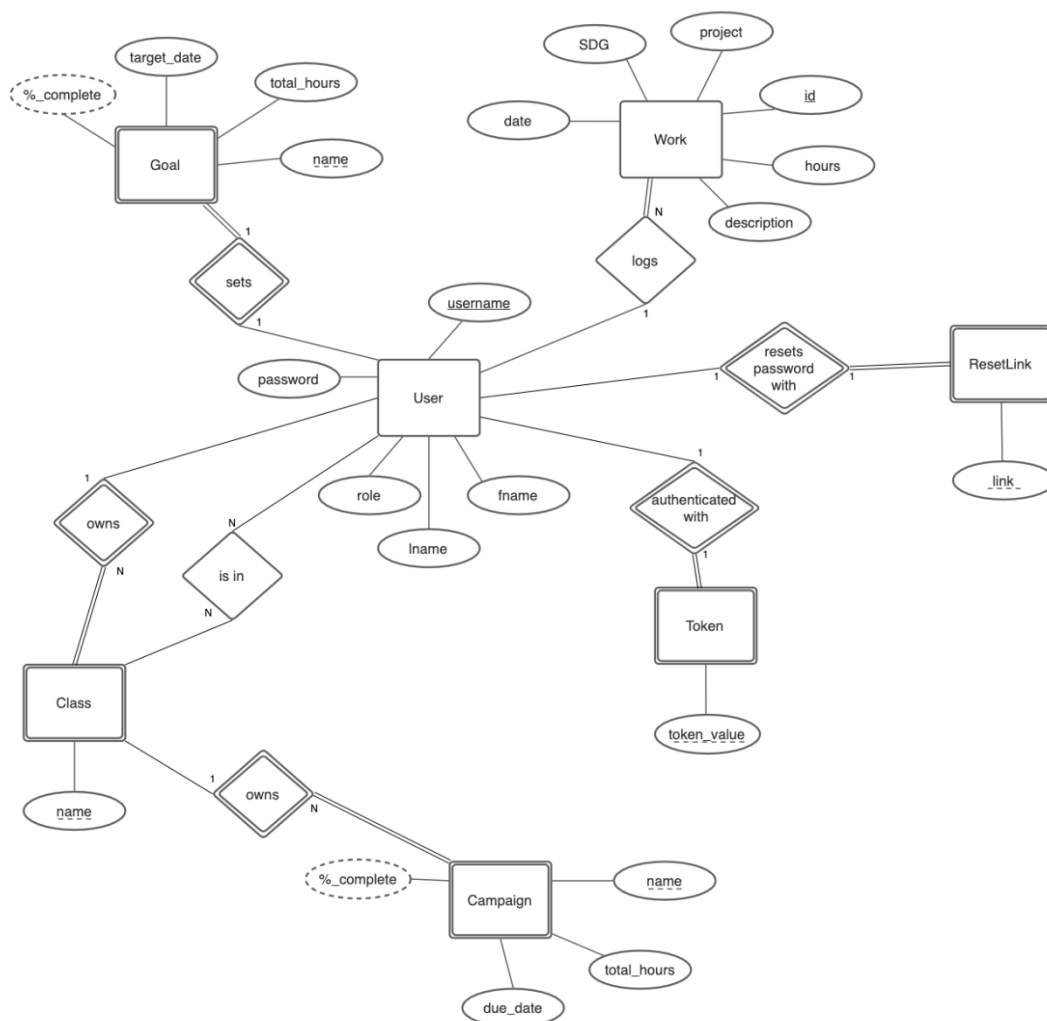


Figure 3.1: Entity Relationship Diagram for S.L.I. App database

3.1.1 Database Schema

Here, we will describe each entity and highlight primary keys (underlined) and foreign key dependencies.

3.1.1.1 Entities

User: (username, password, role, fname, lname)

- The user table is where we store basic login information for each user and the designated role (student or teacher), and if they are a teacher, we store their name.

Class: (teacher [fk1], name)

- This keeps track of all the classes a single teacher owns.

Token: (user [fk2], token_val)

- When users login, they are given a unique token value associated with their username to maintain their session as they travel between pages.

InClass: (teacher, class [fk3], student [fk4])

- This table manages which students are associated with which class

Work: (id, user [fk5], project, SDG, date, hours, description)

- When a student logs work, it is added to the Work table and associated with their account.

Campaign: (teacher, class [fk6], name, total_hours, due_date)

- Teachers create campaigns (or assignments) for their students in the specified class to complete.

Goal: (user [fk7], name, total_hours, target_date)

- Students can create goals which are linked to their account and keep track of their progress.

ResetLink: (user [fk8], link)

- If a teacher forgets their password, a unique link extension is generated that allows them to reset their password for their account.

3.1.1.2 Foreign Keys

The following list denotes the foreign keys listed above:

- fk1: teacher -> User.username
- fk2: user -> User.username
- fk3: (teacher, class) -> Class.teacher, Class.name
- fk4: student -> User.username
- fk5: user -> User.username
- fk6: (teacher, class) -> Class.teacher, Class.name
- fk7: user -> User.username
- fk8: user -> User.username

3.2 File Use

In the application, the only files that are used are image files. These include icons on the users' dashboards and the S.L.I. logo that will be found throughout the app. All of the images used in the app are standardized to be in PNG format.

3.3 Data Exchange

The frontend of the application collects user input and passes it to the backend logic via http POST requests. The integration between the frontend and backend is facilitated through a Restful API. Then, only the backend logic can communicate with the MySQL database to insert, update, retrieve, or delete data.

3.4 Security

Given the nature of the application being centered around a personalized user experience, user login is required to be able to make use of the functionality. When storing user login data, we encrypt the username/email and password (using the Fernet encryption module) so that the database does not hold actual emails and passwords. Also, we only collect personally identifiable information if the user is a teacher. This information includes first and last name and their email address, which is encrypted. The backend logic, which is separate from the database contains the function to decrypt the data to ensure security. In terms of students, no personally identifiable information is stored in order to protect their identities because the main target audience is elementary aged students. We have given teachers the responsibility of creating student accounts, so they are able to create arbitrary usernames and passwords that do not reveal the students' identities.

4 Component Design

4.1 Static Design

The diagram seen below is a static representation of the components in the system, broken down into the four layers. The user interface section shows each individual component, or screen, and each screen points to the API calls that each one makes. Then, each API call maps to a business logic function via a URL generated through the Flask application, which then interact with specific entities in the database through queries executed through the MySQL library. The entities in the MySQL database section represent each table in the database.

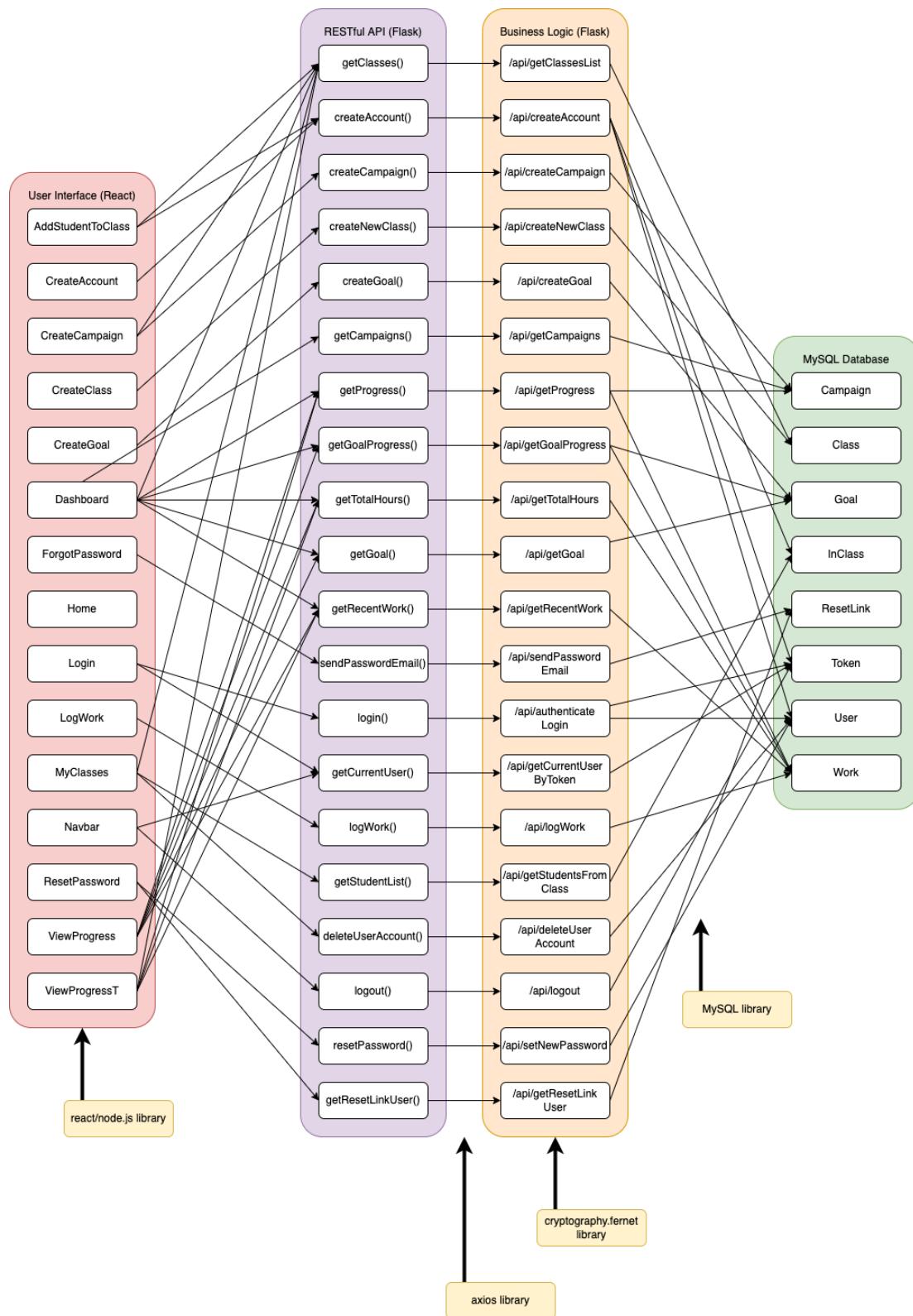


Figure 4.1: Static component diagram for S.L.I. application

4.2 Dynamic Design

The dynamic component design can be seen through the robustness diagram shown below. It shows the runtime interactions between the user and the system during a scenario where a teacher creates an account and sets up her class. The teacher first creates an account which calls the backend API and creates a user in the database. Then, the teacher tries to log in but forgets their password. They access the forgot password page to reset their password. The teacher can then access the dashboard page once they log in. The dashboard allows the teacher to manage their class, create campaigns, log work, and create goals. The API calls are handled by the backend Restful API. To see this diagram in work, image the use case of “teacher create account and setup class”. The teacher would follow the steps explained above. The diagram shows how the use case interacts with our controller and entity classes, as well as other pages from the dashboard.

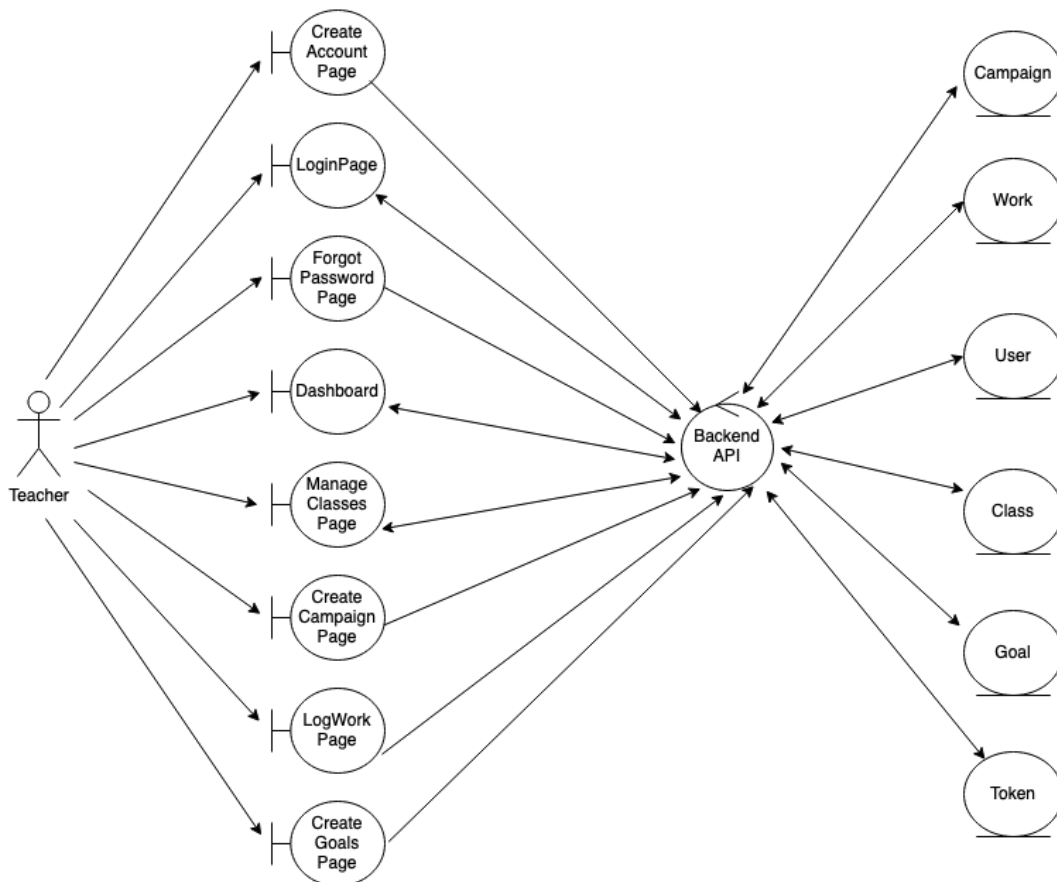


Figure 4.2: Robustness diagram for S.L.I. application

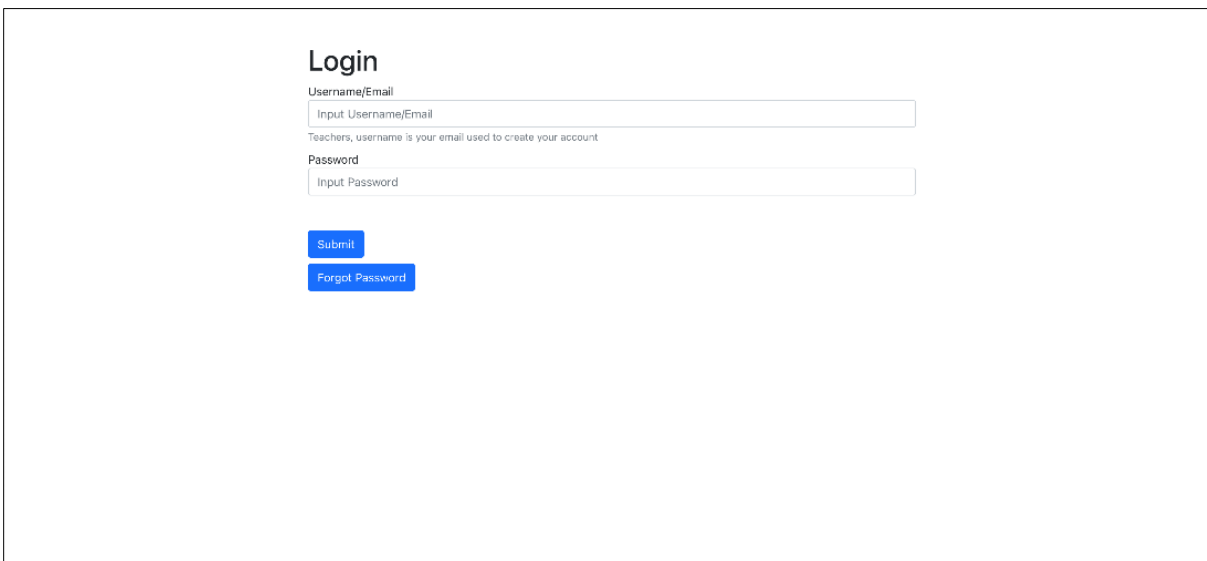
5 UI Design

In this section, we will present the User Interface (UI) portion of the app which students and teachers will use when logging their work or managing classes. We will discuss our UI design decisions and describe all major screens students will interact with.

Upon navigating the web to the application, the user is greeted with the home screen as shown in Figure 6.1. This screen is rather simple, leading the user directly to the colorful Login and Create Account buttons, as these will allow the user to create an account and log in (as shown in Figure 6.2). At the login screen, users are prompted to insert their email address/username (dependent on teacher or student status) and password into the fields provided, with an additional “Forgot Password” option available should it be necessary.



Figure 6.1: Home Screen of the Application



Login

Username/Email

Teachers, username is your email used to create your account

Password

[Submit](#)

[Forgot Password](#)

Figure 6.2: Login Screen of the Application

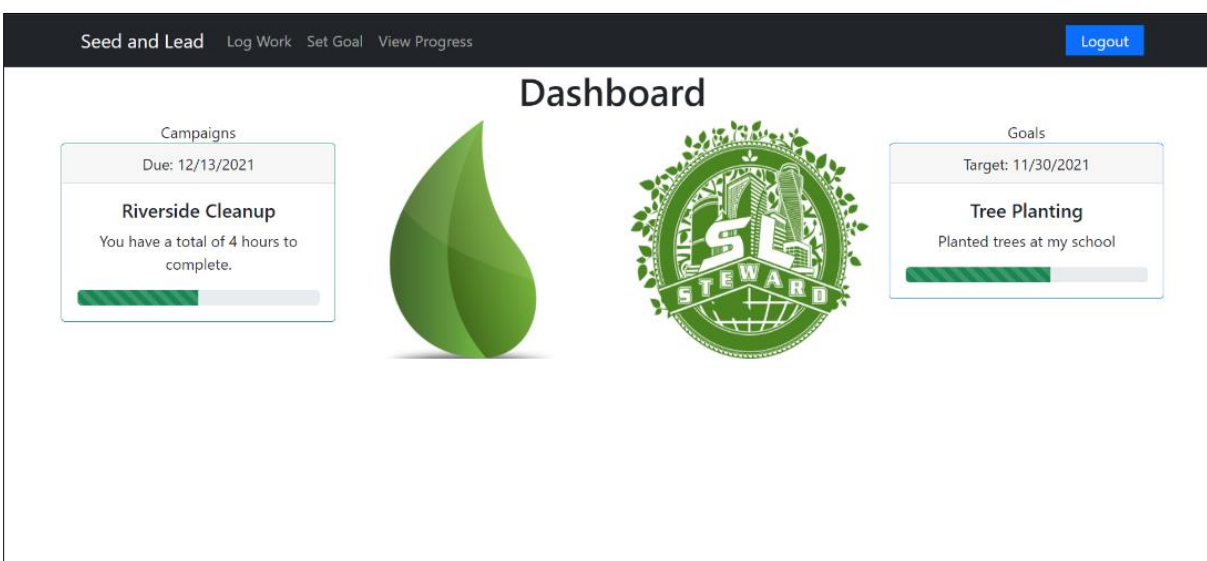


Figure 6.3: Student Dashboard

Once the user submits their login, they are brought to the Dashboard of the app as shown in Figure 5.3. From this Dashboard, users are immediately greeted with their most recent Campaigns and self-set Goal along with visualizations of their progress towards them in the form of green progress bars. A Navigation bar is located at the top of the screen, with all of the major functionalities of the app linked as well as a Logout button, and the “Seed and Lead” button which takes the user back to the dashboard from any of the other pages.

Figure 6.4: Student Log Work Page

A student is able to get to the Log Work page via the Navigation bar at the top of the screen. Once here, a student can input a project they have been working on, including name, date, description of the project, how many hours were completed, and a corresponding SDG for the project. Upon clicking submit, the student will be taken back to the Dashboard where their Campaigns and Goals will be updated to reflect the progress made.

Figure 6.5: Student Set Goal Page

Upon reaching the Set Goal page, students are able to set a number of hours they wish to complete by a certain target date. When clicking submit, the student will be taken

back to the Dashboard and their Goal will be displayed on the right-hand side of the screen.

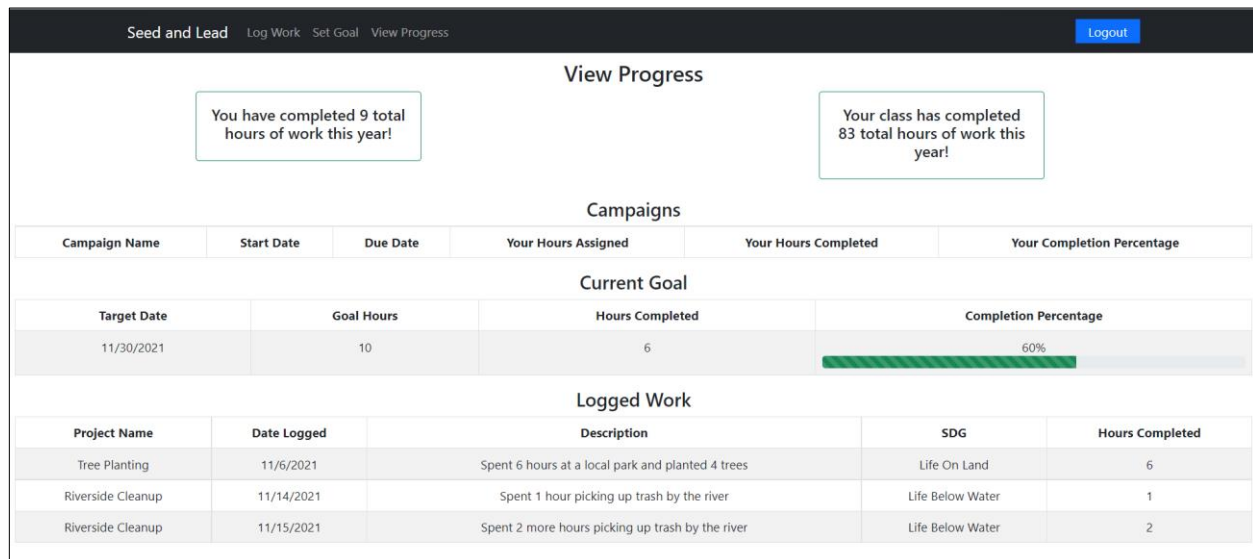


Figure 6.6: Student View Progress Page

The student's View Progress page is where the students are able to see the more descriptive versions of the work they have completed. At the top, students can see the total number of hours they have completed, as well as the total number of hours their class has completed. In the first table, the Campaigns are displayed. Compared to the Dashboard, the start date is additionally displayed, as well as the hours completed towards the campaign. In the second table, the student's Current Goal can be seen with the start date also added, and also the number of hours completed and those wished to be completed. Finally, the last table on the bottom shows general Logged Work for the student. When a student logs work on the Log Work page, their input is transferred here so their work is able to be consolidated into one screen.

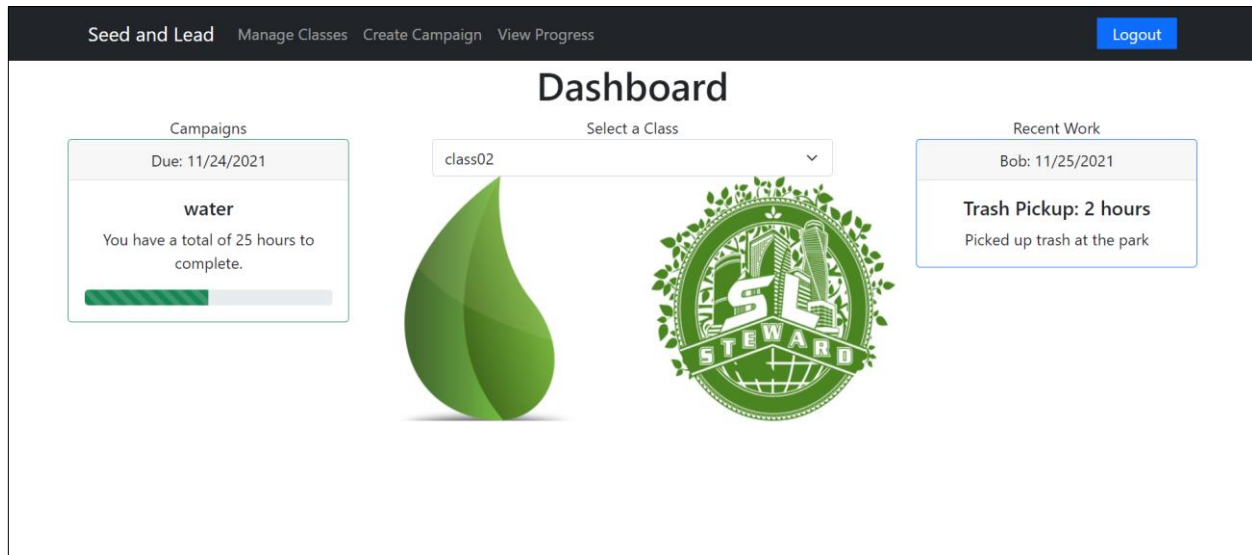


Figure 6.7: Teacher Dashboard

Similar to the Student Dashboard, the Teacher Dashboard is seen in Figure 6.7. The Campaigns are displayed the same way as to the Student, but Teachers have the ability to select a class and see the Campaigns they have created for each class, as well as a “Recent Work” section on the right side of the Dashboard which will display recent logged work that Students have done based on the class selected.

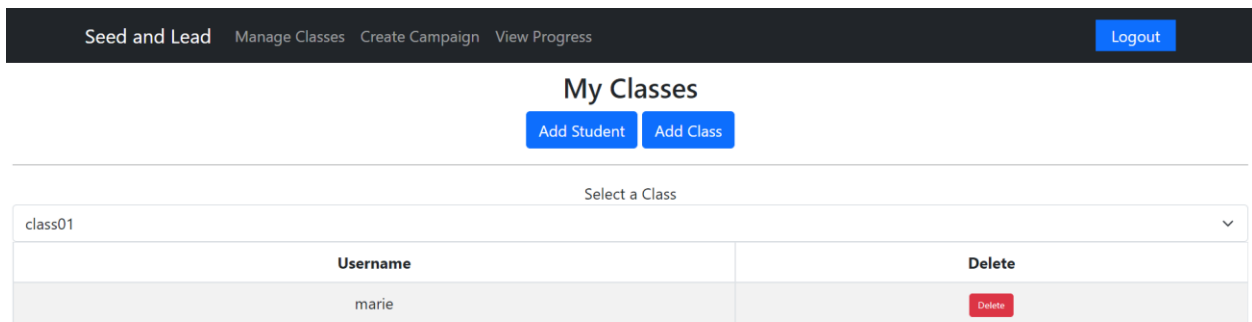
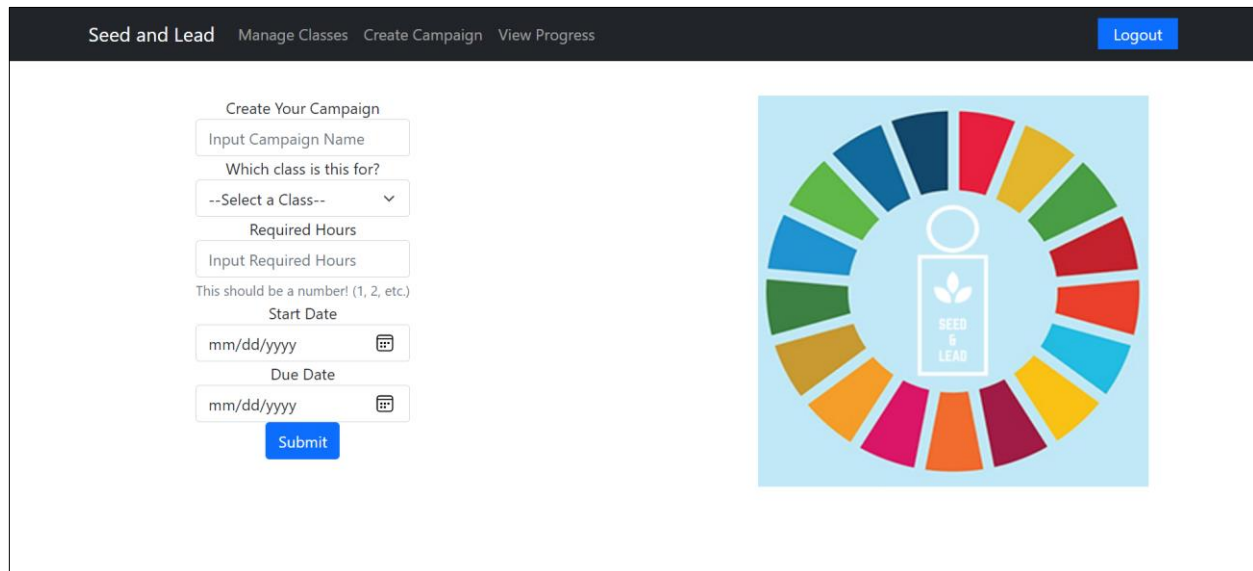


Figure 6.8: Teacher Manage Classes

As seen in Figure 6.8 above, Teachers can navigate to the Manage Classes page via the navigation bar where they are able to create classes or add students to existing

classes. Teachers are able to select a class and view the students in each class, where they are also able to remove the student from the class, if needed.



The screenshot shows the 'Seed and Lead' Teacher Create Campaign interface. At the top, there is a navigation bar with links: 'Seed and Lead', 'Manage Classes', 'Create Campaign', and 'View Progress'. A 'Logout' button is located in the top right corner. The main content area is divided into two sections. On the left, there is a form titled 'Create Your Campaign' with the following fields: 'Input Campaign Name', 'Which class is this for?' (a dropdown menu with '--Select a Class--' as the placeholder), 'Required Hours' (a text input field), 'Input Required Hours' (a text input field), 'Start Date' (a date picker with the format 'mm/dd/yyyy'), and 'Due Date' (a date picker with the format 'mm/dd/yyyy'). A 'Submit' button is at the bottom of the form. On the right, there is a circular graphic with 12 colored segments (blue, green, yellow, red, orange, pink, purple, blue, green, yellow, red, orange) surrounding a central logo that reads 'SEED & LEAD'.

Figure 6.9: Teacher Create Campaign

Teachers are also able to create campaigns for their students which show up on both the Student and Teacher Dashboard. On the page, Teachers create a name for the campaign, select a class which they would like to assign it to, as well as the number of required hours and the start and end dates for the campaign. Upon clicking submit, the Teacher will be taken back to the Dashboard where the Campaign will be displayed when clicking the desired class, as well as will be displayed to the Students on their Dashboard in the desired class.

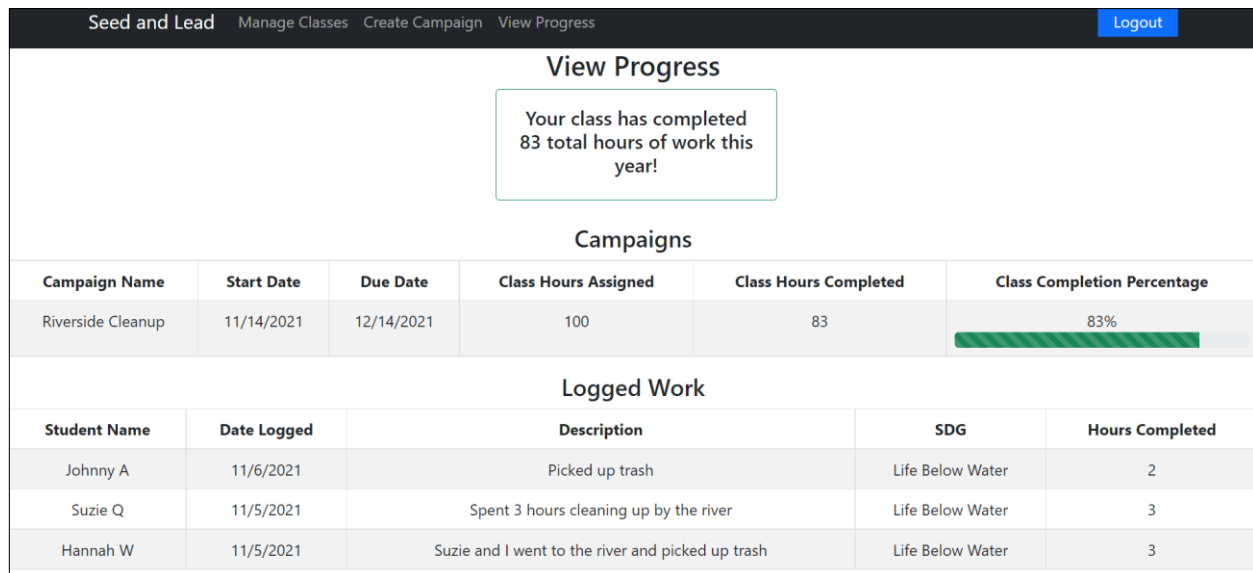


Figure 6.10: Teacher View Progress Page

Similar to the Student View Progress Page, the Teacher View Progress Page can be seen above in Figure 6.10. At the top, Teachers can select the class they want to see progress for, and it will display the total number of hours that have been logged for the desired class. Below that the Campaigns table is displayed which features the Campaigns the Teacher assigned for the class, as well as the hours completed for the class so far. Finally at the bottom, the Logged Work table can be seen which lets the Teacher view all logged work for students in the class, whose values are generated when students log work via the Log Work Page in the Student view.

6 Appendix A: RESTful API Documentation

6.1 Response Codes

Code	Text	Description
200	OK	Success!
201	Created	The request succeeded, and a new resource was created as a result.
400	Bad Request	The server could not understand the request due to invalid syntax.
401	Unauthenticated	The request requires user authentication.
404	Not Found	The server cannot find the requested resource because the URL is not recognized.
500	Internal Server Error	The server experienced an internal error that is beyond the scope of the user.
503	Service Unavailable	The server is not ready to handle the request because it may be overloaded or under maintenance.

POST /api/authenticateLogin

Authenticates and logs user in if username and password input is correct.

Resource URL

<http://127.0.0.1:5000/api/authenticateLogin>

Request Information

Request format: JSON

Resource Information

Response Format: JSON

Requires Authentication: No

Rate Limited: No

Parameters

Parameter	Description
username	Username of user logging in, email if teacher
password	Password of user logging in
role	Role of user logging in (Teacher or Student)

Example Request

<http://127.0.0.1:5000/api/authenticateLogin>

JSON:

```
{
  "username": "my_email@gmail.com",
  "password": "my_password",
  "role": "T"
}
```

Example Response

```
{
  "code": 200,
  "token": "MINCtyRPYx3mJqnuFYZsgd7YuJ4TDqP0",
  "username": "my_email@gmail.com",
  "role": "T",
  "isLoggedIn": True
}
```

Response Fields

Field	Description
code	Response code from request
token	Token generated for user's session
username	Username of user logged in
role	Role of user logged in
isLoggedIn	Whether or not user was successfully logged in

GET /api/getCurrentUserToken

Gets user information and login status to be used for session and on dashboard from a given token.

Resource URL

`http://127.0.0.1:5000/api/getCurrentUserToken?token=<token>`

Resource Information

Response Format: JSON

Requires Authentication: No

Rate Limited: No

Parameters

Parameter	Description
token	Token of user who is logged in, None if no user logged in

Example Request

`http://127.0.0.1:5000/api/getCurrentUserToken?token=MINCtyRPYx3mJqnuFYZsgd7YuJ4TDqP0`

Example Response

```
{
  "code": 200,
  "isLoggedIn": True,
  "username": "my_email@gmail.com",
  "fname": "Bob"
}
```

Response Fields

Field	Description
code	Response code from request
isLoggedIn	Whether or not user is currently logged in
username	Username of user in question
fname	First name of user to be displayed on dashboard

GET /api/getClassesList

Gets a list of classes that a given teacher owns.

Resource URL

<http://127.0.0.1:5000/api/getClassesList?username=<username>>

Resource Information

Response Format: JSON

Requires Authentication: Yes

Rate Limited: No

Parameters

Parameter	Description
username	Username of teacher whose classes

Example Request

http://127.0.0.1:5000/api/getClassesList?username=my_email@gmail.com

Example Response

```
{
  "code": 200,
  "classes": [
    [
      "class01"
    ],
    [
      "class02"
    ]
  ]
}
```

Response Fields

Field	Description
code	Response code from request
classes	List of class names that the teacher owns

GET /api/getStudentsFromClass

Gets a list of student usernames who are in a given teacher's class.

Resource URL

`http://127.0.0.1:5000/api/getStudentsFromClass?className=<className>&teacher=<teacher>`

Resource Information

Response Format: JSON

Requires Authentication: Yes

Rate Limited: No

Parameters

Parameter	Description
className	Specific class whose list of students we want
teacher	Username of teacher who owns the class

Example Request

`http://127.0.0.1:5000/api/getStudentsFromClass?className=class01&teacher=my_email@gmail.com`

Example Response

```
{
  "code": 200,
  "studentList": [
    [
      "student_username01"
    ],
    [
      "student_username02"
    ]
  ]
}
```

Response Fields

Field	Description
code	Response code from request

studentList	List of student username lists that are in the specified class
-------------	--

POST /api/createAccount

Creates a student or teacher account with given login information and logs them in.

Resource URL

<http://127.0.0.1:5000/api/createAccount>

Request Information

Request format: JSON

Resource Information

Response Format: JSON

Requires Authentication: No

Rate Limited: No

Parameters

Parameter	Description
fname	First name of new user if a teacher
lname	Last name of new user if a teacher
username	Username of new user, email if teacher
password	Password of new user
conf_password	Confirmation of password of new user
role	Role of user logging in (Teacher or Student)

Example Request

<http://127.0.0.1:5000/api/createAccount>

JSON:

```
{
  "fname": "George",
  "lname": "Burdell",
  "username": "gburdell01@gmail.com",
  "password": "new_password",
  "conf_password": "new_password"
  "role": "T"
}
```

Example Response

```
{  
  "code": 201,  
  "token": " sLqjkhbDjt7TgRLTb0MsH7FxFxL0hcv1UJ",  
  "username": "gburdell01@gmail.com",  
  "role": "T",  
  "isLoggedIn": True  
}
```

Response Fields

Field	Description
code	Response code from request
token	Token generated for new user's session
username	Username of new user logged in
role	Role of new user logged in
isLoggedIn	Whether or not new user was successfully logged in

POST /api/createNewClass

Creates a new class that the given teacher will own.

Resource URL

<http://127.0.0.1:5000/api/createNewClass>

Request Information

Request format: JSON

Resource Information

Response Format: JSON

Requires Authentication: Yes

Rate Limited: No

Parameters

Parameter	Description
username	Last name of new user if a teacher
className	Username of new user, email if teacher

Example Request

<http://127.0.0.1:5000/api/createNewClass>

JSON:

```
{
  "username": "my_email@gmail.com",
  "className": "new_class"
}
```

Example Response

```
{
  "code": 201
}
```

Response Fields

Field	Description
code	Response code from request

POST /api/sendPasswordEmail

Creates a unique link extension and sends a URL to the teacher for them to reset their password.

Resource URL

<http://127.0.0.1:5000/api/sendPasswordEmail>

Request Information

Request format: JSON

Resource Information

Response Format: JSON

Requires Authentication: No

Rate Limited: No

Parameters

Parameter	Description
email	Email of teacher who is resetting password

Example Request

<http://127.0.0.1:5000/api/sendPasswordEmail>

JSON:

```
{
  "email": "my_email@gmail.com"
}
```

Example Response

```
{
  "code": 201
}
```

Response Fields

Field	Description
code	Response code from request

GET /api/getResetLinkUser

Gets the username associated with the given encrypted password reset link to change the proper password.

Resource URL

`http://127.0.0.1:5000/api/getResetLinkUser?link=<link>`

Resource Information

Response Format: JSON

Requires Authentication: No

Rate Limited: No

Parameters

Parameter	Description
link	Encrypted link associated with user who is resetting password

Example Request

`http://127.0.0.1:5000/api/getResetLinkUser?link=gAAAAABhjuG3P8XrpgWzoB86oCjgP
-
mjPIVoOkJTbcga_Zt_c8BWN2TckeCjzEvRCBvJ_F4ldS94Uz0AA4vuKyP9lI3nzhuM_1C
ALwiU0E4M027xaCdAJ0J2PJ8uERfjzyoS2Yv12Hla`

Example Response

```
{  
  "code": 200,  
  "username": "my_email@gmail.com"  
}
```

Response Fields

Field	Description
code	Response code from request
username	Username of user whose password is being reset

POST /api/logWork

Logs work for a specific student to count towards campaigns and goals.

Resource URL

<http://127.0.0.1:5000/api/logWork>

Request Information

Request format: JSON

Resource Information

Response Format: JSON

Requires Authentication: Yes

Rate Limited: No

Parameters

Parameter	Description
user	Username of user logging work
project	Name of work being logged
SDG	SDG that the work applies to
date	Date that work was completed
hours	Total hours of work
description	Short description of work completed

Example Request

<http://127.0.0.1:5000/api/logWork>

JSON:

```
{
  "user": "student_username01",
  "project": "Recycling",
  "SDG": "Responsible Consumption and Production",
  "date": "2021-11-10",
  "hours": 2,
  "description": "I went with my family to a community recycling event put on by my neighborhood."
}
```

Example Response

```
{  
  "code": 201  
}
```

Response Fields

Field	Description
code	Response code from request

POST /api/logout

Logs the user out and removes session information from database.

Resource URL

http://127.0.0.1:5000/api/logout

Request Information

Request format: JSON

Resource Information

Response Format: JSON

Requires Authentication: Yes

Rate Limited: No

Parameters

Parameter	Description
username	Username of user logging out

Example Request

http://127.0.0.1:5000/api/logout

JSON:

```
{
  "username": "my_email@gmail.com"
}
```

Example Response

```
{
  "code": 201
}
```

Response Fields

Field	Description
code	Response code from request

GET /api/getCampaigns

Gets a list of campaigns assigned to a specific student or a teacher's class depending on the role of the user.

Resource URL

`http://127.0.0.1:5000/api/getCampaigns?role=<role>&username=<username>`

Resource Information

Response Format: JSON

Requires Authentication: Yes

Rate Limited: No

Parameters

Parameter	Description
role	Role of user requesting campaign list
username	Username of user requesting campaign list

Example Request

`http://127.0.0.1:5000/api/getCampaigns?role=T&username=my_email@gmail.com`

Example Response

```
{
  "code": 200,
  "campaignList": [
    [
      "campaign01",
      5,
      2021-11-6,
      2021-11-14
    ],
    [
      "campaign02",
      10,
      2021-11-6,
      2021-11-21
    ]
  ]
}
```

Response Fields

Field	Description
code	Response code from request
campaignList	List of campaigns for specific student or teacher's class in the form [name, total hours, start date, due date]

GET /api/getGoal

Gets a student's goal that they have set for themselves.

Resource URL

`http://127.0.0.1:5000/api/getGoal?username=<username>`

Resource Information

Response Format: JSON

Requires Authentication: Yes

Rate Limited: No

Parameters

Parameter	Description
username	Username of user requesting their goal

Example Request

`http://127.0.0.1:5000/api/getGoal?username=student_username01`

Example Response

```
{
  "code": 200,
  "goal": [
    [
      15,
      2021-11-30
    ]
  ]
}
```

Response Fields

Field	Description
code	Response code from request
goal	List of the student's goal in the form [total hours, target date]

POST /api/createCampaign

Creates a campaign for a specific class.

Resource URL

<http://127.0.0.1:5000/api/createCampaign>

Request Information

Request format: JSON

Resource Information

Response Format: JSON

Requires Authentication: Yes

Rate Limited: No

Parameters

Parameter	Description
teacher	Username of teacher creating campaign
className	Name of class to assign campaign to
name	Name of campaign to be created
hours	Total hours of the campaign
start_date	Start date of the campaign
due_date	Due date of the campaign

Example Request

<http://127.0.0.1:5000/api/createCampaign>

JSON:

```
{
  "teacher": "my_email@gmail.com",
  "className": "class01",
  "name": "campaign03",
  "hours": 10,
  "start_date": 2021-11-22,
  "due_date": 2021-11-18
}
```

Example Response

```
{  
  "code": 201  
}
```

Response Fields

Field	Description
code	Response code from request

POST /api/createGoal

Sets a goal for the given student. Overwrites existing goal if a goal already exists for the student.

Resource URL

<http://127.0.0.1:5000/api/createGoal>

Request Information

Request format: JSON

Resource Information

Response Format: JSON

Requires Authentication: Yes

Rate Limited: No

Parameters

Parameter	Description
username	Username of student setting goal
total_hours	Total hours of the goal
target_date	Target date of the goal

Example Request

<http://127.0.0.1:5000/api/createGoal>

JSON:

```
{
  "username": "student_username02",
  "hours": 10,
  "target_date": 2021-11-30
}
```

Example Response

```
{
  "code": 201
}
```

Response Fields

Field	Description
-------	-------------

code	Response code from request
------	----------------------------

POST /api/setNewPassword

Resets password of a give teacher.

Resource URL

http://127.0.0.1:5000/api/setNewPassword

Request Information

Request format: JSON

Resource Information

Response Format: JSON

Requires Authentication: Yes

Rate Limited: No

Parameters

Parameter	Description
username	Username of teacher resetting password
newPassword	New password for password to be reset to
conf_newPassword	Confirmation of new password

Example Request

http://127.0.0.1:5000/api/setNewPassword

JSON:

```
{
  "username": my_email@gmail.com,
  "newPassword": "my_new_password",
  "conf_newPassword": "my_new_password"
}
```

Example Response

```
{
  "code": 201
}
```

Response Fields

Field	Description
code	Response code from request

GET /api/getProgress

Gets progress towards all campaigns for a particular student or class.

Resource URL

`http://127.0.0.1:5000/api/getProgress?role=<role>&username=<username>&class=<class>[&student_filter=<student_filter>]`

Resource Information

Response Format: JSON

Requires Authentication: Yes

Rate Limited: No

Parameters

Parameter	Description
role	Role of user requesting progress
username	Username of user requesting progress
class	If user is a teacher, name of class whose progress we want
student_filter	Optional, if user is a teacher, username of student whose progress we want

Example Request

`http://127.0.0.1:5000/api/getProgress?role=T&username=my_email@gmail.com&class=class01`

Example Response

```
{
  "code": 200,
  "progress": [
    [
      "campaign01",
      5,
      2021-11-6,
      2021-11-14
    ],
    [
```

```

        [
            "student01",
            3,
            "60%"
        ],
        [
            "student02",
            3,
            "60%"
        ]
    ],
    [
        [
            "campaign02",
            10,
            2021-11-6,
            2021-11-21
        ],
        [
            [
                "student01",
                3,
                "30%"
            ],
            [
                "student02",
                3,
                "30%"
            ]
        ]
    ]
]
}

```

Response Fields

Field	Description
-------	-------------

code	Response code from request
progress	List of student's or all students in class' progress towards all campaigns in the form [[campaign1, progress1], [campaign2, progress2], ...], with campaign# in the form [class name, campaign name, total hours, start date, due date] and progress# in the form [[student1], [student2], ...] with student# in the form [username, hours complete, percentage complete]

GET /api/getTotalHours

Gets total hours of work logged by a student or class within an optional time frame.

Resource URL

`http://127.0.0.1:5000/api/getTotalHours?username=<username>&role=<role>&class=<class>[&student_filter=<student_filter>][&start_date=<start_date>][&end_date=<end_date>]`

Resource Information

Response Format: JSON

Requires Authentication: Yes

Rate Limited: No

Parameters

Parameter	Description
username	Username of user requesting total hours
role	Role of user requesting total hours
class	If user is a teacher, name of class whose total hours we want
student_filter	Optional, if user is a teacher, username of student whose total hours we want
start_date	Optional, beginning date threshold to calculate total hours
end_date	Optional, ending date threshold to calculate total hours

Example Request

`http://127.0.0.1:5000/api/getTotalHours?username=student_username01&role=S`

Example Response

```
{
  "code": 200,
  "total_hours": [
    8
  ]
}
```

Response Fields

Field	Description
code	Response code from request
total_hours	List of total hours logged for the given student or class

GET /api/getRecentWork

Gets recently logged or all work for a specific student or class.

Resource URL

`http://127.0.0.1:5000/api/getRecentWork?username=<username>&role=<role>&class=<class>&all_work=<all_work>`

Resource Information

Response Format: JSON

Requires Authentication: Yes

Rate Limited: No

Parameters

Parameter	Description
username	Username of user requesting recently logged work
role	Role of user requesting recently logged work
class	If user is a teacher, name of class whose recently logged work we want
all_work	Boolean flag to signal if we want all logged work or only past 14 days

Example Request

`http://127.0.0.1:5000/api/getTotalHours?username=student_username02&role=S&all_work=False`

Example Response

```
{
  "code": 200,
  "recent_work": [
    [
      "student_username02",
      "Planting trees",
      "Life on Land",
      "2021-11-5",
      2,
      "I spent time with my church planting trees in the local park."
    ]
  ]
}
```

```

    ],
    [
        "student_username02",
        "Picking up trash from river",
        "Clean Water and Sanitation",
        2021-11-12,
        3,
        "My family went down to the river in my
        neighborhood and picked up trash."
    ]
]
}

```

Response Fields

Field	Description
code	Response code from request
recent_work	If there is recent work, list of recent work items logged by the given student or class in the form [username, project name, SDG, date, hours, description]

GET /api/getGoalProgress

Gets a student's progress towards their goal.

Resource URL

`http://127.0.0.1:5000/api/getRecentWork?username=<username>`

Resource Information

Response Format: JSON

Requires Authentication: Yes

Rate Limited: No

Parameters

Parameter	Description
username	Username of user requesting recently logged work

Example Request

`http://127.0.0.1:5000/api/getTotalHours?username=student_username02`

Example Response

```
{
  "code": 200,
  "total_hours": 10,
  "current_hours": 5
}
```

Response Fields

Field	Description
code	Response code from request
total_hours	Total hours in the goal
current_hours	Current hours completed towards the goal

POST /api/deleteUserAccount

Deletes a student's account.

Resource URL

http://127.0.0.1:5000/api/deleteUserAccount

Request Information

Request format: JSON

Resource Information

Response Format: JSON

Requires Authentication: Yes

Rate Limited: No

Parameters

Parameter	Description
currStudent	Username of student whose account is being deleted

Example Request

http://127.0.0.1:5000/api/setNewPassword

JSON:

```
{  
  "currStudent": student_username02  
}
```

Example Response

```
{  
  "code": 201  
}
```

Response Fields

Field	Description
code	Response code from request