Criterion C: Development

Techniques used to Create the Program:

1.) Inheritance
2.) Model/View/Controller Architecture
    a.) Encapsulation
    b.) Polymorphism
3.) Searching for specified data in file
4.) Login Page
5.) CRUD Functionality (Create, Read, Update, and Delete functions)
    a.) Inheritance
    b.) Encapsulation
    c.) Polymorphism
    d.) REST API (Representational State Transfer Application Programming Interface)
    e.) Dynamic Data Structure

Imports and Dependencies:

This program uses the following dependencies:

- net.imagej
- jackson-databind
- thymeleaf-layout-dialect
- mysql-conector-java
- spring-boot-starter-jdbc
- spring-boot-starter-web
- spring-boot-starter-validation
- spring-boot-devtools
- spring-boot-starter-test
- spring-boot-testcontainers
- junit-jupiter
- spring-orm

Inheritance through Time objects

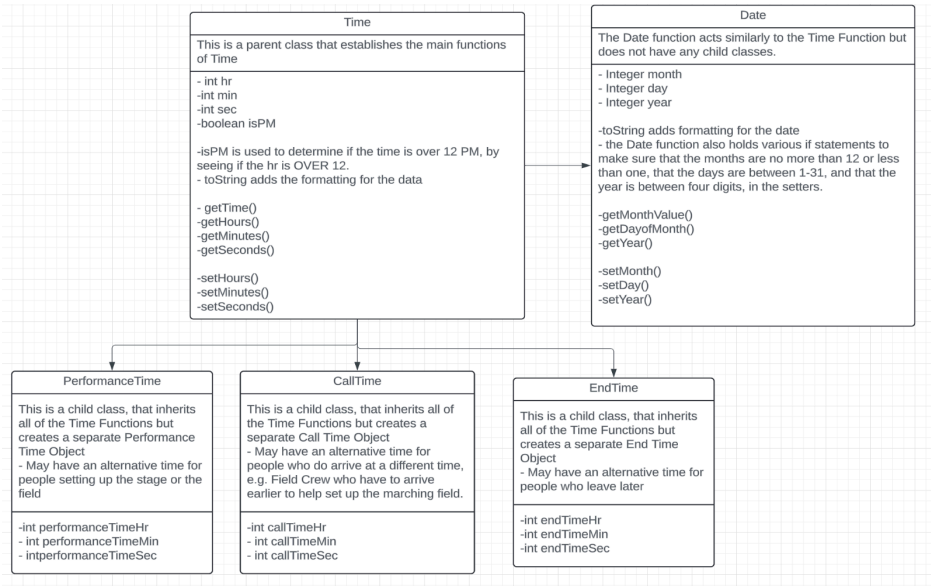

Figure 1: UML Diagram of Time and Date Functions

Going into this project, I understood that users would need to know three different times: the time students are expected to arrive, the performance time, and also the time the event is expected to end. Rather than create multiple “Time” Objects, I decided to create one parent “Time” class. Then I created

three child time classes that **inherit** the features of the Time object. Functionally, all of these objects operate with three integer properties that contain getter and setter methods that can be accessed in other layers of the project through **encapsulation**.

Model/View/Controller (MVC) Project Design

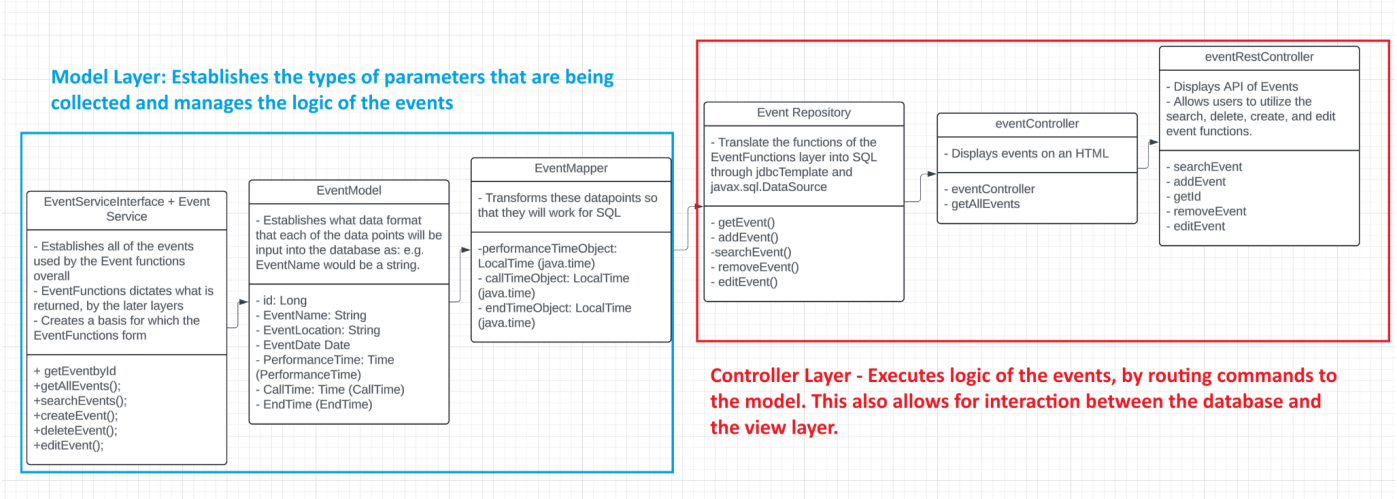

Figure 2: Model and Controller Layers

My program is based in the Model/View/Controller architecture, with three layers acting as the Model layer and three other layers acting as a Controller Layer. The model layers establish the general logic of the application by determining what actions can be performed,what parameters are collected, and interacting directly with the database.

The EventModel() is responsible for defining parameters in event object such as the Event Name.
All of these aspects are then condensed into a single eventModel() object that can access all aspects of the eventModel().

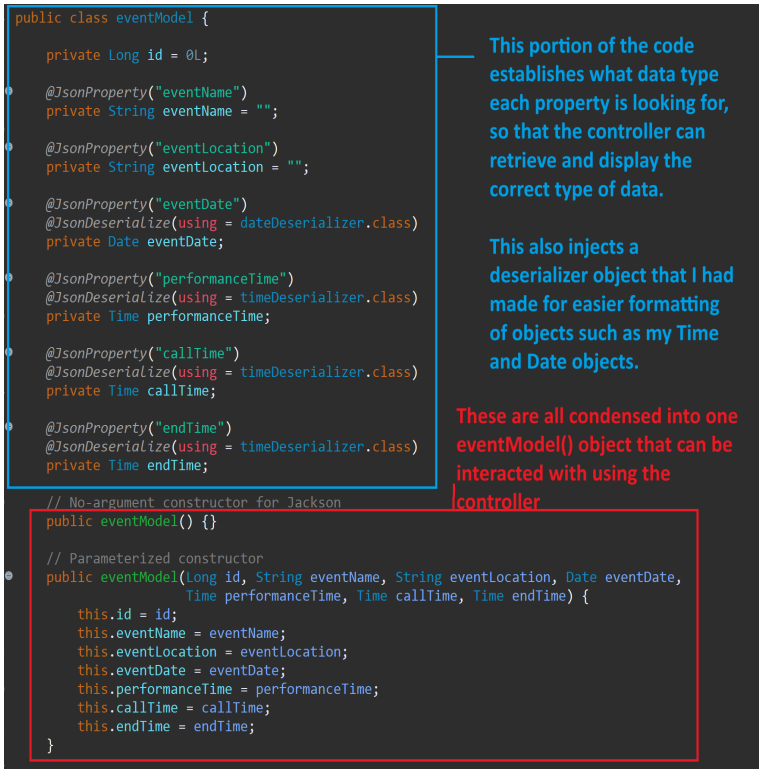

Figure 3: Description of the eventModel

The eventModel() also demonstrates **encapsulation** in which all of the properties of the event are private. Encapsulation allows one to use different properties of the eventModel() depending on user needs. The EventService() establishes the functions used by the repository methods in the controller methods, smoothening the transaction between the repository and the controller with business logic.

```
@Override
public List<eventModel> getAllEvents() {
    // TODO Auto-generated method stub
    List<eventModel> events = shamrockregiment.getEvent();
    System.out.println("Fetched events: " + events);  // Add logging
    return events;


}

@Override

public List<eventModel> searchEvents(String searchEvent) {
    return shamrockregiment.searchEvent(searchEvent);

}

@Override
@Transactional
public eventModel createEvent(eventModel newEvent) {
    System.out.println("Service received event: " + newEvent);

    eventModel savedEvent = shamrockregiment.createEvent(newEvent);

    System.out.println("Event saved with ID: " + savedEvent.getId());
    return savedEvent;
}


@Override
public boolean deleteEvent(long id) {
    return shamrockregiment.removeEvent(id);
}

@Override
public eventModel editEvent(long idToEdit, eventModel editedEvent) {
    return shamrockregiment.editEvent(idToEdit, editedEvent);
}
```

Figure 4: EventService() functions

My EventService() function uses **polymorphism**, in which methods like my eventRestController() accept EventService() methods. This allows for the method to use and extend upon existing EventService() methods. The eventRestController() receives HTTP requests when specific buttons are pressed, injecting the information that it receives from the model and service layer directly into the front-end. The eventController() functions similarly, but injects the information from the eventModel(), so that the getAllEvents() is called, causing all events to be displayed.

```
@Controller
@RequestMapping("/events")


public class eventController {

    private final EventServiceInterface eventService;

    @Autowired

     public eventController(@Qualifier("EventService") EventServiceInterface eventService) {
        this.eventService = eventService;
    }

    @GetMapping("/")

public String getAllEvents(Model model) {


    List<eventModel> events = eventService.getAllEvents();

        model.addAttribute("eventList", events);
        model.addAttribute("title", "Upcoming Shamrock Regiment and Colorguard Events!");

        System.out.println("Events Retrieved"+ events);
        return "events";
    }
```

Figure 5: eventController() function

REST API controllers were used to simplify communication between the front and back end, using Java Spring's @RestController, giving me end points that send HTTP requests to JSON. The needs of the front end and back end are able to work together, but are further separated with the REST API.

```
@RestController
@RequestMapping("/api/v1/event")


public class eventRestController {

    @Autowired
    @Qualifier("EventFunctions")
    private final EventServiceInterface service;
    public eventRestController(EventServiceInterface service) {
        this.service = service;
    }

    @GetMapping("/")
    public List<eventModel> displayEvents() {

    List<eventModel> events = service.getAllEvents();


        return events;

    }

    @GetMapping("/search/{searchEvent}")

    public List<eventModel> searchEvent(@PathVariable(name = "searchEvent") String searchEvent) {


        List<eventModel> events = service.searchEvents(searchEvent);
        return events;

    }
```

Figure 6: eventRestController()

Search Functions

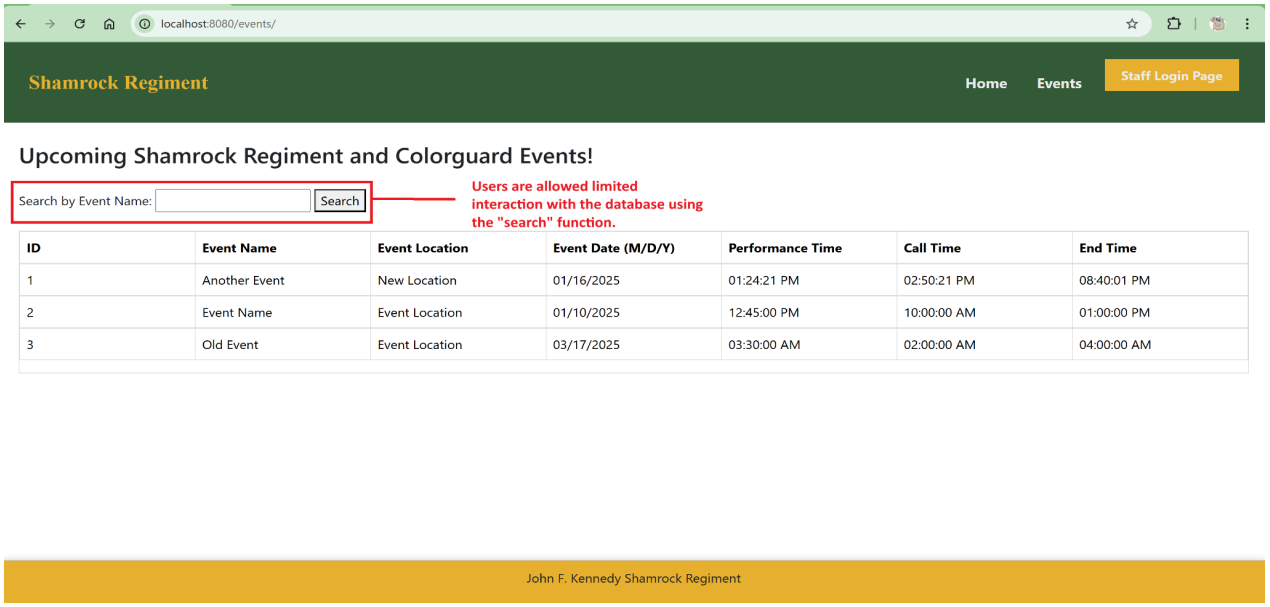

Figure 7: /event function without searching

When the events.html page is opened, the user can view a basic layout of existing events. Users are limited in their ability to use use the database, and are able to “search” by event name.

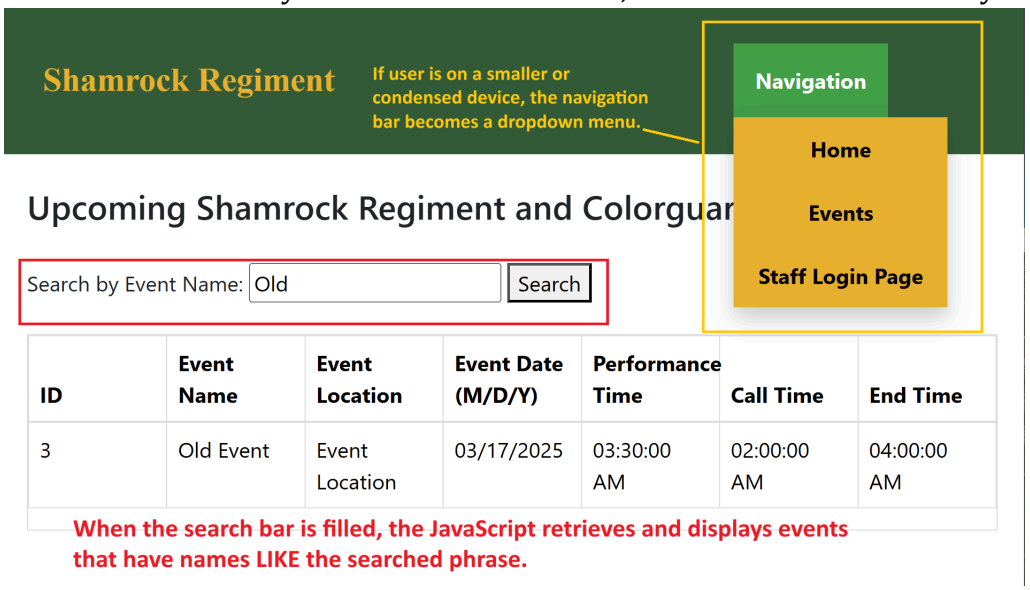

Figure 8: events.html page AFTER searching, condensed

Search requests are submitted through the Javascript file shamrockRegimentStaff.js to change input from the form into SQL.

```
function searchEvent(event) {
    event.preventDefault(); // Stop default form submission

    const eventName = document.getElementById("eventName").value.trim();

    if (!eventName) {
        alert("Please enter an event name to search.");
        return;
    }

    fetch(`/api/v1/event/search/${encodeURIComponent(eventName)}`)
        .then(response => response.json())
        .then(events => {
            updateEventTable(events);
        })
        .catch(error => {
            console.error("Error searching events:", error);
        });
}
```

Figure 9: Javascript for the search button (searchEvent)

```
@GetMapping("/search/{searchEvent}")

public List<eventModel> searchEvent(@PathVariable(name = "searchEvent") String searchEvent) {


    List<eventModel> events = service.searchEvents(searchEvent);
    return events;

}
```

Figure 10: Java for eventRestController()

The request is submitted to the eventRestController() which directs the request and the information to the repository layer.

```
@Override
public List<eventModel> searchEvent(String searchEvent) {
    //Logging to make sure that the console is searching for the CORRECT term
    System.out.println("Search Results: " + searchEvent);

    //Replacing the placeholder null value with the searchEvent string
    String query = "SELECT * FROM `event` WHERE EventName LIKE ?";
    String finalQuery = query.replace("?", "'%" + searchEvent.trim() + "%'");
    System.out.println("Executing query: " + finalQuery);

    //Altering the database to reflect the results
    List<eventModel> results = jdbcTemplate.query(query, new eventsMapper(),
            "%" + searchEvent.trim() + "%");

    //Returning SIMILAR terms to the front-end
    return results;
}
```

Figure 11: Java for the searchEvent function in the eventRepository

The repository layer submits this information into the database as a SELECT query, which is then submitted back into the Javascript layer to display data that is LIKE the submitted request.

```
function updateEventTable(events) {
    const tableBody = document.querySelector("table tbody");

    // Clear existing table rows
    tableBody.innerHTML = "";

    if (events.length === 0) {
tableBody.innerHTML = `<tr><td colspan="7" class="text-center">No events found.</td></tr>`;
        return;
    }

    // Populate table with search results
    events.forEach(event => {
        const row = `<tr>
            <td>${event.id}</td>
            <td>${event.eventName}</td>
            <td>${event.eventLocation}</td>
            <td>${formatDate(event.eventDate)}</td>
            <td>${formatTime(event.performanceTime)}</td>
            <td>${formatTime(event.callTime)}</td>
            <td>${formatTime(event.endTime)}</td>
        </tr>`;
        tableBody.insertAdjacentHTML("beforeend", row);
    });
}
```

Figure 12: JavaScript for updating the event table

The updateEventTable function returns the results of the query, populating the table with search results.

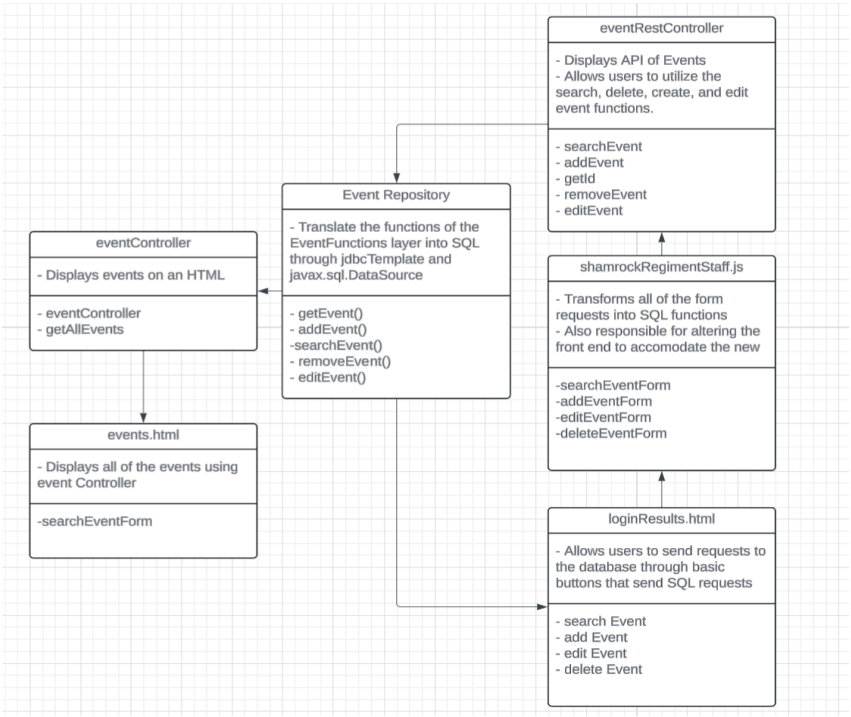

Figure 13: UML Diagram of the EventRepository, the eventController, the eventRestControllers, and the HTML pages.

This demonstrates the basic relationship between the controller functions and between the View functions.

Login Model and Login Controller:

Users may interact with the database depending on their authority within the database.

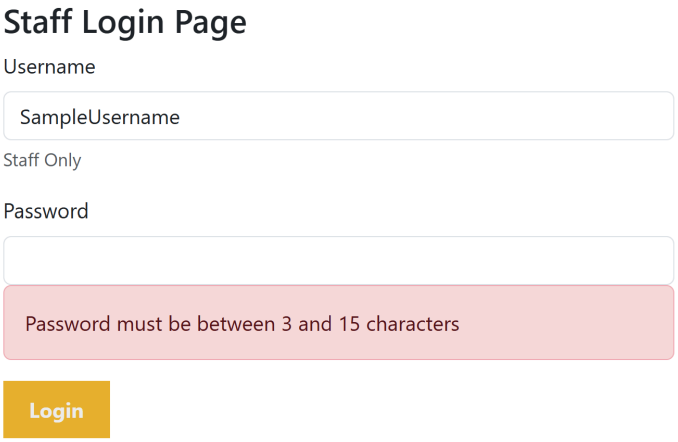

Figure 14: The error message displayed for a password that is too short for the Staff Login Page

Though the login page does not utilize database functionality, my client stressed the importance of limiting user access to secure event information.

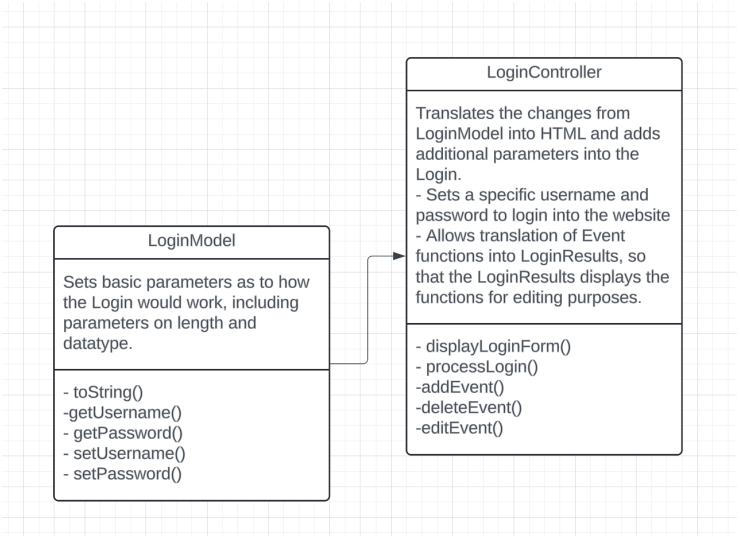

Figure 15: UML Diagram of LoginModel and LoginController function

Following a basic Model/View/Controller structure, the loginController() establishes the correct username as “AdminUser” and the correct password as “UserPassword”.

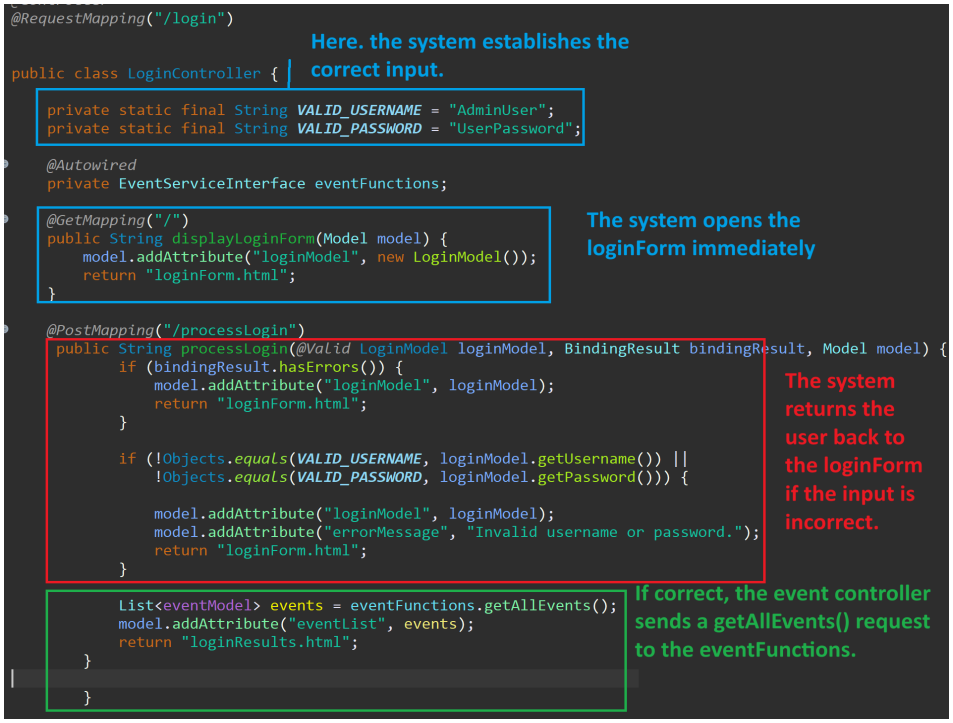

Figure 16: Explanation of loginController page

Requests for loginForm.html is opened immediately when /login page is opened. User attempts to login are sent to the controller as a request, with the controller returning an error message if login information is incorrect. Successfully logging in sends the login controller a getAllEvents() request to eventFunctions() to display existing events alongside additional buttons for database interaction.

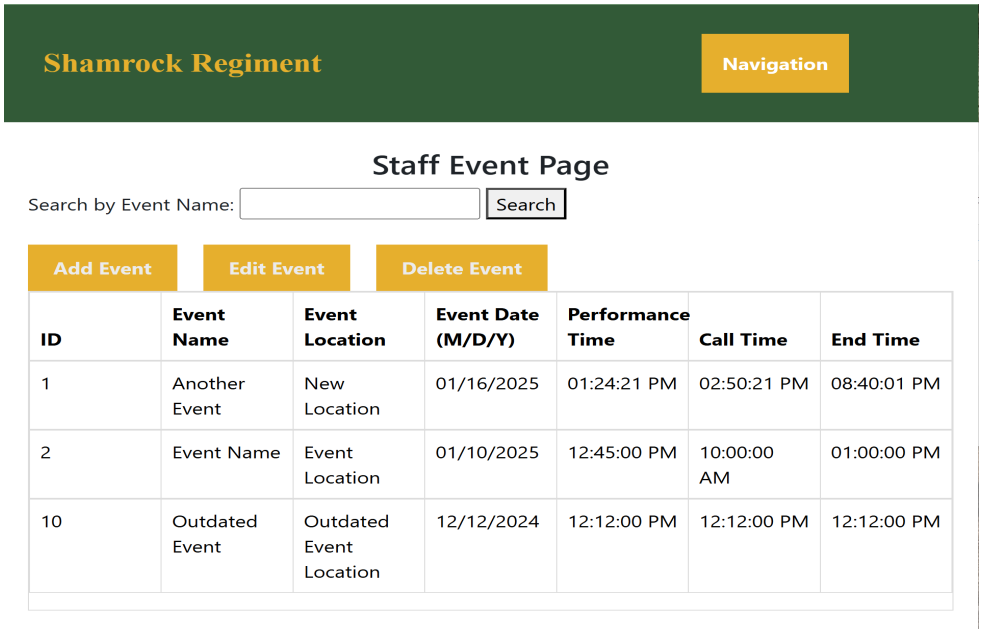

Figure 17: Staff Event Page after successful login

Create, Read, Update, and Delete (CRUD) functionality:

The search function acts as the READ function in a CRUD based database. Java specializes in CRUD functionality, and in using CRUD functions, users are able to interact easily with the database regardless of technical ability. The “Add Event” button acts as this system’s Create function.

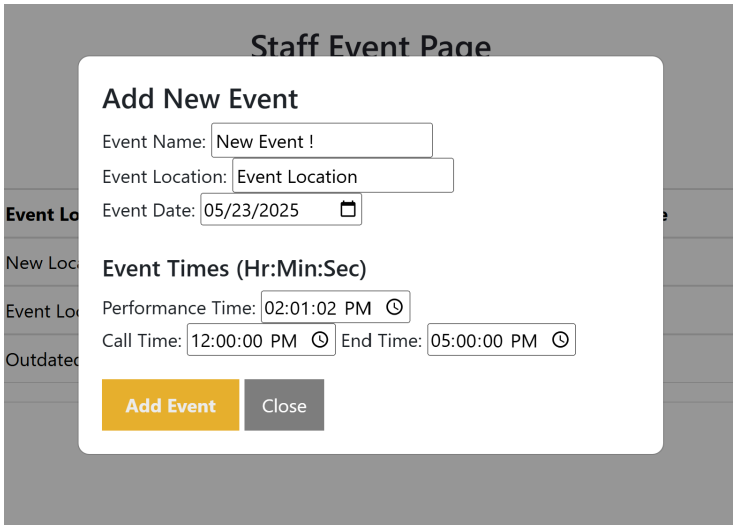

Figure 18: Add Event Modal with Event Information Implemented

After the “Add Event” button is clicked, the user is shown a modal that asks for event information input. The input is transformed with JavaScript in the shamrockRegimentStaff.js into the appropriate data types and variables. This data is then packaged into a string containing the data (eventData), sent into the eventRestController(), which sends it as a request for the eventRepository to submit to the database.

Figure 19: Javascript to submit input from add event button

```
@PostMapping(value = "/create", consumes = "application/json")
public eventModel createEvent(@RequestBody eventModel newEvent) {
    System.out.println("Received POST request to create event: " + newEvent);

    return service.createEvent(newEvent);


}
```
Figure 20: eventRestController() createEvent

The repository layer then sets up the query to be submitted to the database, and reformats the time objects into the correct format.

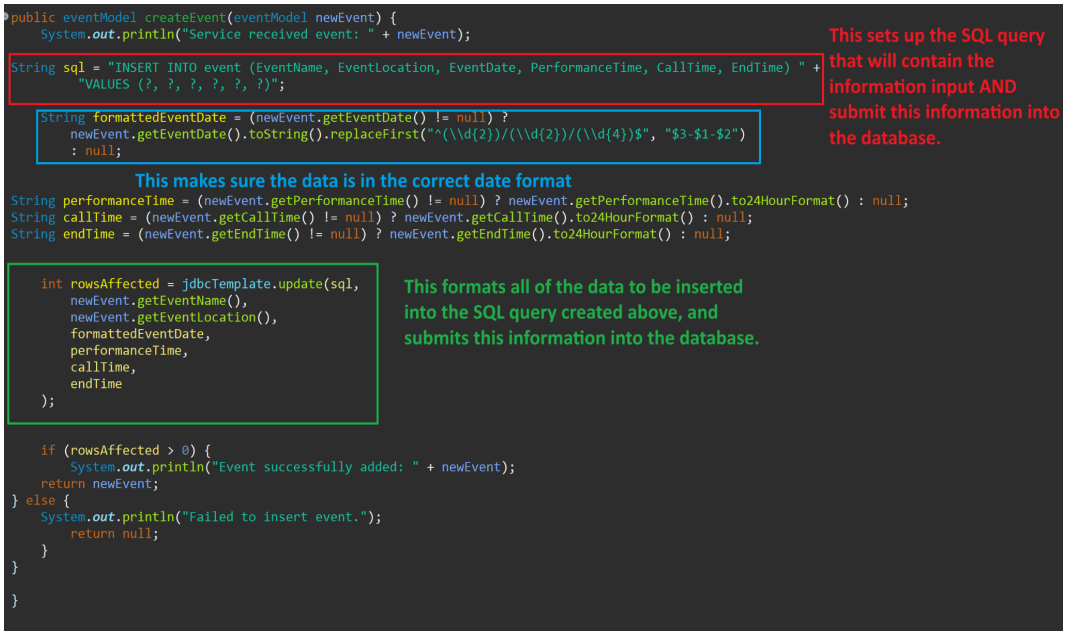

Figure 21: eventRepository layer submitting a request to the database to create a new event

Then, it submits all of the information collected in the database and when the user reloads the page the new event appears in the database and in the front-end.

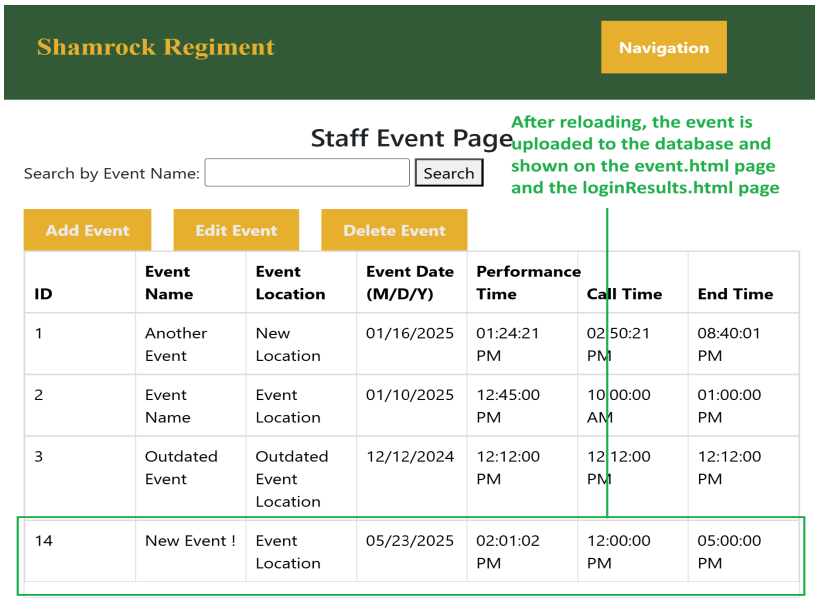

Figure 22: loginResults.html after adding an event

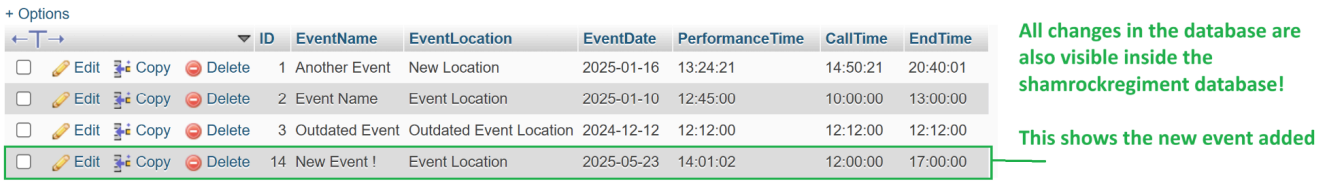

Figure 23: shamrockRegiment database after adding an event using the addEvents() button.

The Edit Event button functions acts as an update function in this application, in which the system presents a modal for the user to input the event ID.

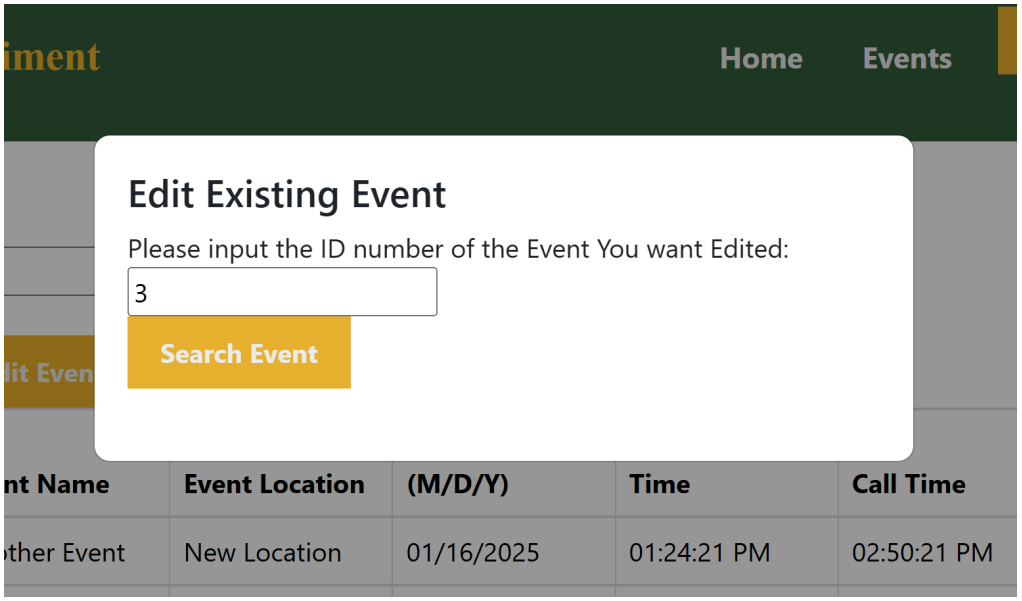

Figure 24: ID input portion of my Edit Event modal.

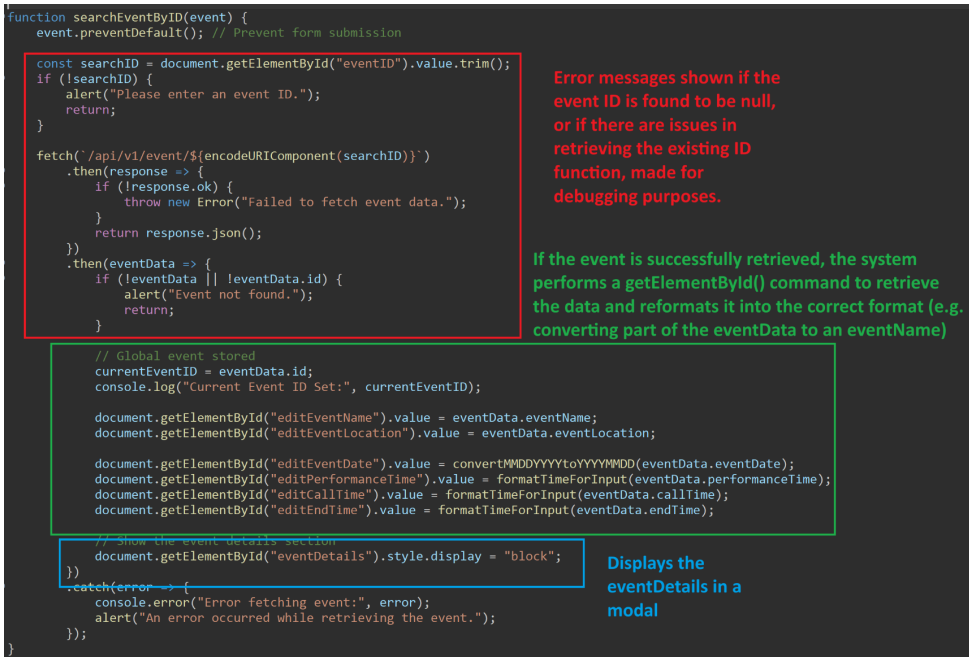

Figure 25: searchEventByID JavaScript

This input is retrieved from the shamrockRegimentStaff.js JavaScript, and converted into the correct format for the modal to retrieve the event. This data is then sent to the modal for the user to see the current information and edit the information on the page.

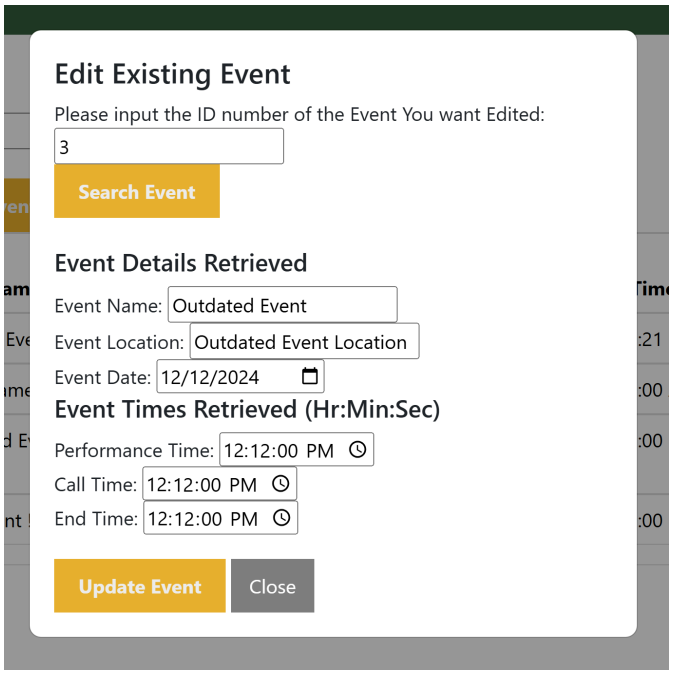

Figure 26: Expanded Edit Event modal

After editing, the new information will be retrieved from the shamrockRegimentStaff.js through the submitEditEventForm(), to be sent into the eventRestController().

Figure 27: submitEditEventForm()

The eventRestController() then sends the “POST” request to the eventRepository, to send all of the information input to the actual database.

```
@PutMapping("/edit/{id}")

public eventModel edit(@RequestBody eventModel model, @PathVariable(name="id") long id) {
    System.out.println("Received Edit Request for ID:" + id);
    System.out.println("Updated Event Data:" + model.toString());
    return service.editEvent(id,model);
    }
```

Figure 28: eventRestController() request.

The eventRepository reformats this data as an SQL query to be sent to the database. This reformats the data, and changes both the database and the front end.

```java
@Override
public eventModel editEvent(long idToEdit, eventModel editEvent) {
    System.out.println("Editing event with ID: " + idToEdit);
    System.out.println("Updated event data: " + editEvent);

    // Ensure consistent date format (yyyy-MM-dd)
    String formattedEventDate = (editEvent.getEventDate() != null) ?
        editEvent.getEventDate().toString().replaceFirst("^(\\d{2})/(\\d{2})/(\\d{4})$", "$3-$1-$2")
        : null;

    // Ensure times are in 24-hour format
    String performanceTime = (editEvent.getPerformanceTime() != null) ? editEvent.getPerformanceTime().to24HourFormat() : null;
    String callTime = (editEvent.getCallTime() != null) ? editEvent.getCallTime().to24HourFormat() : null;
    String endTime = (editEvent.getEndTime() != null) ? editEvent.getEndTime().to24HourFormat() : null;

    // Execute update query
    int result = jdbcTemplate.update(
        "UPDATE `event` SET EventName = ?, EventLocation = ?, EventDate = ?, PerformanceTime = ?, CallTime = ?, EndTime = ? WHERE ID = ?",
        editEvent.getEventName(),
        editEvent.getEventLocation(),
        formattedEventDate,
        performanceTime,
        callTime,
        endTime,
        idToEdit
    );

    // Check if update was successful
    if (result > 0) {
        editEvent.setId(idToEdit);  // Ensure ID remains unchanged
        System.out.println("Event successfully updated: " + editEvent);
        return editEvent;
    } else {
        System.out.println("Failed to update event with ID: " + idToEdit);
        return null;
    }
}
```

Sends all of the collected and reformatted event information into the database as an SQL "UPDATE" query.

Figure 29: eventRepository sending an UPDATE request to the database.

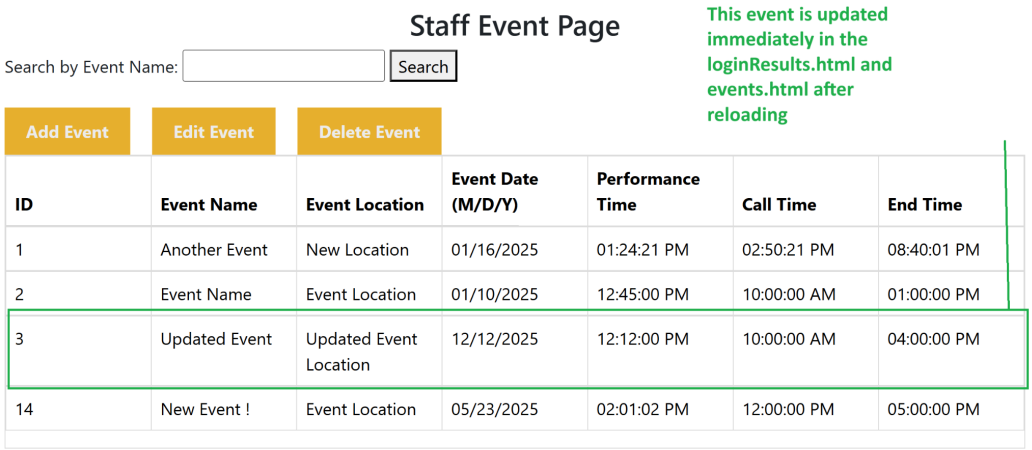

Figure 30: Staff Event Page after uploading

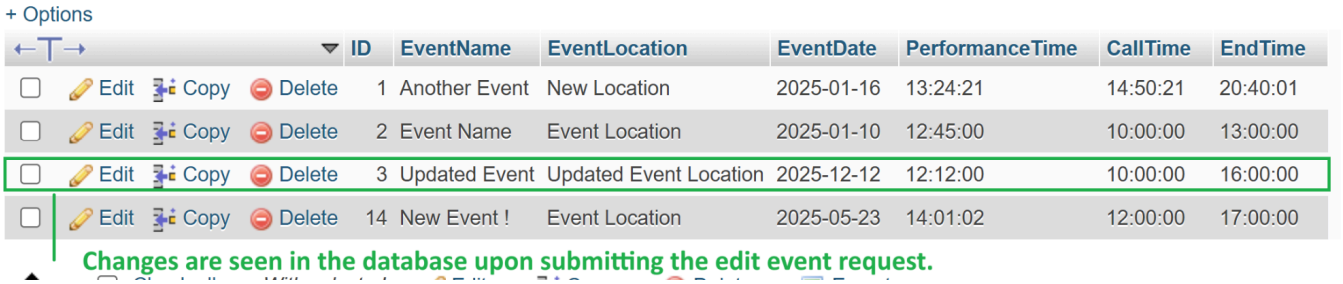

Figure 31: events table after uploading

As for the “delete event” button, the following modal is displayed to retrieve the ID of the event being deleted:

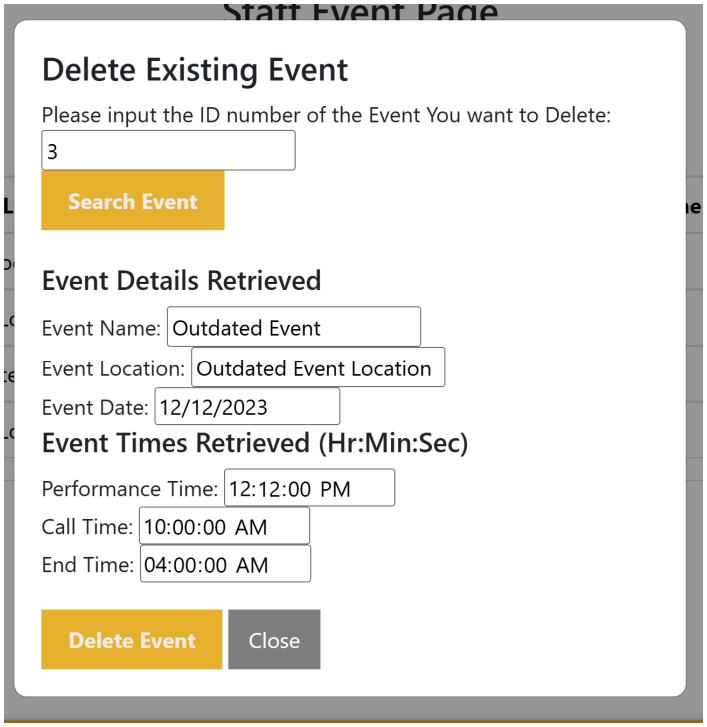

Figure 32: Delete Event modal after ID information is retrieved.

The “GET” function performed to retrieve all of the event information functions exactly the same in the delete function, but has different variables to allow the user to view the event information. The differences in the functionality is found within the submission of the delete event requests to the database.

The initial request is first confirmed through shamrockRegimentStaff.js, and when it is successful, it sends that request to the eventRestController() as a simple ‘DELETE’ function.

```
function confirmDeleteEvent() {
    const eventId = document.getElementById("deleteEventID").value.trim();
    if (!eventId) {
        alert("Please enter a valid event ID.");
        return;
    }

    if (window.confirm("Are you sure you want to delete this event?")) {

        deleteEvent(eventId);
    }
}

async function deleteEvent(eventId) {
    try {
        const response = await fetch(`/api/v1/event/remove/${eventId}`, {
            method: 'DELETE'
        });

        if (response.ok) {
            alert("Event deleted successfully!");
            closeDeleteModal();

        } else {
            const errorMessage = await response.text();
            alert(`Failed to delete event. Reason: ${errorMessage}`);
        }
    } catch (error) {
        console.error("Error deleting event:", error);
        alert("An error occurred while deleting the event.");
    }
}
```

This sends an error message if the ID input is incorrect or is unable to be retrieved.

If the input is correct, this sends a confirmation alert, which then directs the code to the deleteEvent function.

This function then sends a request to the eventRestController() as a DELETE method.

Figure 33: Javascript of deleteEvent function

This then sends *only* the ID into the database for the SQL to reference in the eventRepository, since the ID is a property that is unique for each event and is determined by the database.

```
@DeleteMapping("/remove/{id}")

public boolean removeEvent(@PathVariable(name="id") long id) {
    return service.deleteEvent(id);
}
```

Figure 34: eventRestController() sending the ID to the eventRepository.

The eventRepository then sends the ID to the eventRepository, where it is referenced in an SQL 'DELETE' query. After a successful deletion, the database and the front end should *not* have the event information displayed.

```
@Override
public boolean removeEvent(long id) {
    int result = jdbcTemplate.update("DELETE FROM `event` WHERE ID = ?", id );

    if (result > 0) {
        return true;
    }
    else {
        return false;
    }

}
```

This simply retrieves the ID that is collected from the database, and submits a DELETE query.

Figure 35: eventRepository sending a delete request

Works Cited

"Discover What Encapsulation Is | Definition and Overview." *Sumo Logic*, https://www.sumologic.com/glossary/encapsulation/. Accessed 7 Apr. 2025.

"Lucidchart | Diagramming Powered By Intelligence." *Lucidchart*, https://www.lucidchart.com/pages/landing. Accessed 7 Apr. 2025.

MVC - MDN Web Docs Glossary: Definitions of Web-Related Terms | MDN. 20 Dec. 2023, https://developer.mozilla.org/en-US/docs/Glossary/MVC.

Sluiter, Shad."Web Development in Java Spring Boot." *YouTube*, http://www.youtube.com/playlist?list=PLhPyEFL5u-i0MHw41OapxjLlsl0s2XQyE. Accessed 7 Apr. 2025.

What Is a REST API? https://www.redhat.com/en/topics/api/what-is-a-rest-api. Accessed 7 Apr. 2025.

"What Is Polymorphism? Definition, Types, & Examples." *Sumo Logic*, https://www.sumologic.com/glossary/polymorphism/. Accessed 7 Apr. 2025.