CSCU9YW Report

2710761

# Problem

The task is to design a web service for an events registration business, pulling user information from an external service and managing event storage and user association. The design should follow REST principles, while catering to two different user types: normal users, who can view, register and unregister themselves for events, and administrators, who can manage events and view attendees for events.

## Assumptions

* The web service shouldn’t permanently store User details, other than the User ID. There is no reason to have multiple copies of users’ details when they can be retrieved from the user details API.
* Users register for events with just their user id, and don’t need to login with a password. As we cannot access actual user accounts from the API, there would be no way to implement this even if necessary.
* Users can register for any number of events, but cannot register for the same event twice (unless they unregister, then re-register again)
* Any number of events can be created

## Implemented Functionality

|  |
| --- |
| **For Users:** |
| View a list of all events |
| View all events the user is registered for |
| View more detail about a specific event (location, date, time, etc.) |
| Register for a specific event using their ID |
| Unregister for a specific event using their ID |
| **For Administrators:** |
| View all attendees to an event |
| Delete an event |
| Unregister a specific user from an event |
| Create a new event |
| **Miscellaneous:** |
| Users validated automatically by the web service, whenever an Event object is request, it will validate its attendees before returning. |
| Secured admin functionality: to access the admin page, the correct admin password must be provided. This password is stored in a session variable, and must be provided when making requests to admin endpoints |
| Properly formatted output for users – using HTML tables, rather than raw JSON |
| A user-friendly interface |

# Implementation

The web service itself is built using Java, implementing the Spring Web and Spring Security libraries, while the user-facing frontend is constructed with HTML, JavaScript, CSS and making a lot of use of the Bootstrap library to use some pre-constructed components and styles. The client and server communicate through HTTP Requests.

The HTTP Requests sent by the client are handled by our Controller class, using the appropriate endpoint as well as any necessary parameters, such as user or event IDs. Using these requests clients can access event details, register, and unregister for events, and view all events they are currently registered to. Administrators can also create and delete events, as well as view the list of attendees for events. All of the heavy lifting is handled by Spring, so most of our code does not implement anything other than what a novice level developer would consider new, depending on your viewpoint this can be a positive or negative.

## Data Structures

The main data structure in the system is the Event class, which holds all information relating to Events, including attendees. Events are not stored persistently, but a few are statically generated when the web service starts up. Events are not stored persistently as this is a 10-credit module, and my dissertation is a 60-credit module, so I dedicated most time to that instead. Instead, the events are stored in a HashMap data structure, using a unique event ID as the key and the Event object as the value. The HashMap allows a pseudo-replication of how events would be accessed from an actual persistent data structure, through the use of an indexed/primary key (the EventID) rather than through regular searching as would be done through something like an ArrayList. Events hold attributes for the ID, a description of the event (effectively the title), the location, date and time, duration, and maximum capacity, as well as the list of attending users stored as an ArrayList. Users are stored in an ArrayList rather than a HashMap as we do not store the actual User object, rather we only store their unique ID. This follows the principles of data normalisation, by ensuring we don’t have redundant/repeated information stored. It also means we can easier validate users, as whenever all of a Users’ details are required, we request them straight from the API using the stored ID, rather than just using our own stored data which could potentially be outdated.

The Event class has methods for checking if an attendee is attending a specific event, adding, and removing attendees (based on a user id), as well as basic setters for all other attributes. It also has a sanitise method, which returns a replica event without the attendee list – this means the replica object can be passed to unauthorised clients without exposing the list of attending users’ details to them.

The events can be accessed through any implementation of the EventService interface, using the implemented getters.

## REST Principles

The design of the system follows basic REST principles as is the industry standard. As resources are all stored using a unique id number, it constitutes having its own URI, which allows for identification of resources. Requests are made using JSON, this is particularly handy as both Java and JavaScript provide easy parsing between JSON and class Objects. The web service is also stateless, meaning that it does not store any data about the client’s session – this is stored on the client itself.

Caching has not been implemented to the service, but it would be logical and beneficial to do so. Storing the results from GET requests (such as the list of all events) would increase performance and reduce load on the server, while still providing all the required functionality.

Finally, the web service implements a uniform and logical interface. To access events, the “/events/{uid}” endpoint is used always, for admin related functions, the “/admin/…” endpoint. This makes it more intuitive for developers to use and understand the API.

For requesting/supplying resources, the client can utilise the following request methods:

* GET Requests – these are used to request a resource from the system, for example getting all events or attendees for an event. Our system requires that the unique ID for the resource be passed as part of URL (e.g. host.com/events/{event id here})
* POST Requests – POSTs are used to submit a resource to the server, such as creating a new event. The content of the resource is submitted through the POST body, which via Spring will be converted into a Java object when received.
* PUT Requests – Used to update or overwrite existing resources but can also be used to create new resources. For consistency, our system only uses it to update resources such as altering the attendees to an event. Usage of PUT requests are the same as POST requests, using a JSON body and serialising to a Java object
* DELETE Requests – As the name implies, DELETE requests will remove a resource from the server. These are only used by requests from the administrator panel, as we would not want any users being able to alter the content on the web server in such a way. Currently the only resource that can be deleted is Events, through their unique ID. DELETE requests can be submitted the same as GET requests, with the unique ID of the resource supplied in the URI.

All of these requests will return a HTTP Status Code indicating the result of the operation, and if necessary, further details in the response headers and body – such as the ID of a newly created event if a POST request was submitted, or the content of a GET request. The application makes use of many status codes, but the most common are the following:

* 200 (OK) – if the operation was completed successfully, and there is nothing else to return
* 400 (Bad Request) if the web service was supplied a malformed URI
* 404 (Not Found) if the web service was supplied a valid format URI, but the ID for the event or user does not exist
* 409 (Conflict) if the client tries to create a resource that already exists, or tries to register to an event they already signed up for, or if they unregister for an event they are not signed up for
* 401 (Unauthorized) if a user tries to access an admin function endpoint with an incorrect or no Authorization header
* 204 (No Content) if the request was successful but no other content needs to be returned, e.g. from a delete request where it would not be possible to return a resource location or ID (as it no longer exists)

## Security

The application secures its sensitive functionality primarily through the use of an administrator password that must be supplied. This password authenticates the client before accessing any administrator pages. Furthermore, if the user is authorised the password will be stored as part of their session and passed in the Authorization header in any admin-related requests (managing events, getting attendees) so that the server can authenticate before returning any data.

The password is first encoded to Base64, which is done to convert any non-http compatible characters into regular characters that can be sent as required. After this it is sent to the server (encrypted via HTTPS), where it is received and compared against the stored password. If the password is correct, the system will return a ‘true’ value, which the client then receives and stores the password for later use (in any Authorization headers to future admin endpoint calls). This system is not secure, the password is stored in plaintext on the server, and in plaintext of the client if the server has validated it (as part of their session variable). It would be safer to return a login or session token back from the server, rather than a simple Boolean. This token could then be sent in the authorization header instead and checked by the server to see if the user was previously validated. This solution also means the server can control factors like the timeout of the users’ login session. Still though, it

Communication between the client and server is also encrypted via HTTPS, this is enabled through Spring and a self-signed certificate.

## Scalability & Reliability

The application is very simple and would scale up easily. The stateless nature of the service means that it would be able to handle huge numbers of users and would likely be bottlenecked by the network connection before the service itself became an issue. If the system had implemented a persistent data structure, such as a database or serialised object storage, accessing the data could slow down responses to users but depending on the implementation of the data structure it could be negligible (e.g., indexed databases are very fast at retrieving data, compared to flat file structures). The service also has good reliability, through testing some bugs were uncovered that would cause the system to fail and not send an appropriate response to the user. These bugs were fixed, and it seems like the vast majority of cases where the system would fail have been fixed (of course, no system is perfect).

# Code Listing

## Event Class

public class Event {  
 private static int *eventCounter* = 0;  
  
 @Id  
 private final int eventId;  
 private final String description;  
 private final String location;  
 private final String date;  
 private final String time;  
 private final String duration;  
 private final int maxCapacity;  
  
 */\*\*  
 \* A HashSet of all user IDs that are attending this event  
 \*/* private ArrayList<String> attendees;  
  
 private Event(int eventId, String description, String location, String date, String time, String duration, int maxCapacity) {  
 this.eventId = eventId;  
 this.description = description;  
 this.location = location;  
 this.date = date;  
 this.time = time;  
 this.duration = duration;  
 this.maxCapacity = maxCapacity;  
  
 attendees = new ArrayList<>();  
 }  
  
 */\*\*  
 \* Generates a new Event object with a unique ID  
 \** ***@param*** *description  
 \** ***@param*** *location  
 \** ***@param*** *date  
 \** ***@param*** *time  
 \** ***@param*** *duration  
 \** ***@param*** *maxCapacity  
 \*/* public Event(@JsonProperty("description") String description, @JsonProperty("location") String location,  
 @JsonProperty("date") String date, @JsonProperty("time") String time,  
 @JsonProperty("duration") String duration, @JsonProperty("maxCapacity") int maxCapacity){  
 this(*eventCounter*++, description, location, date, time, duration, maxCapacity);  
 }  
  
 public Event(Event oldEvent) {  
 this.eventId = oldEvent.eventId;  
 this.description = oldEvent.description;  
 this.location = oldEvent.location;  
 this.date = oldEvent.date;  
 this.time = oldEvent.time;  
 this.duration = oldEvent.duration;  
 this.maxCapacity = oldEvent.maxCapacity;  
  
 attendees = new ArrayList<>();  
 }  
  
 */\*\*  
 \* Check if a user (through their userid) is registered for this event  
 \** ***@param*** *user  
 \** ***@return*** *\*/* public boolean hasAttendee(User user){  
 return hasAttendee(user.getUid());  
 }  
  
 public boolean hasAttendee(String givenUserId){  
 for(String storedUserID : attendees){  
 if(storedUserID.equals(givenUserId)){  
 return true;  
 }  
 }  
  
 return false;  
 }  
  
 */\*\*  
 \* Adds a user to this events attendee list, as long as the capacity isnt reached  
 \* and the user isnt already registered to attend.  
 \** ***@param*** *user  
 \** ***@return*** *True if the user was successfully registered, false if the event is at capacity or if the user  
 \* already registered.  
 \*/* public boolean addAttendee(User user){  
 if(attendees.size() >= maxCapacity){  
 return false;  
 }  
  
 if(hasAttendee(user)) {  
 System.*out*.println("User already registered for event!");  
 return false;  
 }  
  
 attendees.add(user.getUid());  
 System.*out*.println(user + " registered for " + this);  
 return true;  
 }  
  
 */\*\*  
 \* Removes a user from the attending set, or does nothing if they are not attending  
 \** ***@param*** *user  
 \*/* public boolean removeAttendee(User user) {  
 return removeAttendee(user.getUid());  
 }  
  
 public boolean removeAttendee(String userId){  
 attendees.indexOf(userId);  
 int index = -1;  
  
 for(int i = 0; i < attendees.size(); i++){  
 if(attendees.get(i).equals(userId)){  
 index = i;  
 break;  
 }  
 }  
  
 if(index != -1) {  
 attendees.remove(index);  
 System.*out*.println(userId + " unregistered for " + this);  
 return true;  
 }  
  
 return false;  
 }  
  
 */\*\*  
 \* Create a new event with the sensitive data removed (i.e. the attendees list)  
 \** ***@return*** *\*/* public Event sanitise(){  
 return new Event(this);  
 }  
  
  
 public ArrayList<String> getAttendees(){  
 return attendees;  
 }  
  
 public int getAttendeeCount(){  
 return attendees.size();  
 }  
  
 public int getEventId() {  
 return eventId;  
 }  
  
 public String getDescription() {  
 return description;  
 }  
  
 public String getLocation() {  
 return location;  
 }  
  
 public String getDate() {  
 return date;  
 }  
  
 public String getTime() {  
 return time;  
 }  
  
 public String getDuration() {  
 return duration;  
 }  
  
 public int getMaxCapacity() {  
 return maxCapacity;  
 }  
  
 public String toString(){  
 return description + " (" + eventId + ") ";  
 }  
}

## User Class

*/\*\*  
 \* An immutable class representing a User of the system  
 \* (this is also the same as an Attendee)  
 \*/*public class User {  
 @Id  
 private final String uid;  
 private final String name;  
 private final List<String> interests;  
  
 User(String id, String name, List<String> interests){  
 uid = id;  
 this.name = name;  
 this.interests = interests;  
 }  
  
 public String getUid(){  
 return this.uid;  
 }  
  
 public String getName(){  
 return this.name;  
 }  
  
 public List<String> getInterests(){  
 return this.interests;  
 }  
  
 public String toString(){  
 return name + " (" + uid + ") ";  
 }  
}

## Register Submission Class

*/\*\*  
 \* Validates a userId and eventId given through a POST request  
 \*/*public class RegisterSubmission {  
 private User user;  
 private Event event;  
  
 public RegisterSubmission(String userId, String eventId){  
 EventService service = new ServiceImplementation();  
  
 int parsedEventId;  
 try {  
 parsedEventId = Integer.*parseInt*(eventId);  
 }catch(NumberFormatException e){  
 return;  
 }  
 Event event = service.getEvent(parsedEventId);  
 User user = service.getUser(userId);  
  
 if(event == null || user == null){  
 return;  
 }  
  
 this.user = user;  
 this.event = event;  
 }  
  
 public User user() {  
 return user;  
 }  
  
 public Event event() {  
 return event;  
 }  
}

## SecurityConfig Class

@Configuration  
@EnableWebSecurity  
public class SecurityConfig extends WebSecurityConfigurerAdapter{  
 @Override  
 protected void configure(final HttpSecurity http) throws Exception {  
 http  
 .csrf().disable()  
 .authorizeRequests()  
 .antMatchers("/\*\*").permitAll()  
  
 .and()  
 .formLogin()  
 .loginPage("/login.html")  
 .defaultSuccessUrl("/events.html", true);  
 }  
  
 */\*\** ***@Bean*** *SecurityFilterChain filterChain(HttpSecurity http) throws Exception{  
 /\*\*return http  
 .requiresChannel(channel ->  
 channel.anyRequest().requiresSecure())  
 .authorizeRequests(authorize ->  
 authorize.anyRequest().permitAll())  
 .build();  
 }\*\*/* protected void configure(final AuthenticationManagerBuilder auth) throws Exception {  
 auth.inMemoryAuthentication()  
 .withUser("User1").password(passwordEncoder().encode("password1")).roles("USER")  
 .and()  
 .withUser("user2").password(passwordEncoder().encode("password2")).roles("USER")  
 .and()  
 .withUser("Administrator").password(passwordEncoder().encode("123456789")).roles("ADMIN");  
 }  
  
 @Bean  
 public PasswordEncoder passwordEncoder() {  
 return new BCryptPasswordEncoder();  
 }  
}

## EventService Interface

public interface EventService {  
 //User methods  
 boolean authenticate(String uid);  
 User getUser(String uid);  
 List<Event> getUsersEvents(String userId);  
  
 //Events methods  
 int createEvent(Event event);  
 boolean deleteEvent(int eventId);  
 Event getEvent(int id);  
 ArrayList<Event> getAllEvents();  
 ArrayList<Event> getAllEventsSanitized();  
 List<User> getEventsAttendees(int eventId);  
 int getEventAttendeesCount(int eventID);  
 void validateAttendees();  
 boolean registerUser(Event event, User user);  
 boolean unregisterUser(Event event, User user);  
}

## ServiceImplementation Class

@Service  
public class ServiceImplementation implements EventService{  
 private static final String *USER\_DETAILS\_API* = "https://pmaier.eu.pythonanywhere.com";  
 private static HashMap<Integer, Event> *events* = new HashMap<>();  
  
 //Statically generate some hardcoded Events  
 static{  
 *events*.put(0, new Event("Intelligence Through the Ages", "Wallace Monument", "1 April 2025", "9am", "45 minutes", 1234));  
 *events*.put(1, new Event("Internet of Stuff", "University of Stirling", "1 June 2022", "2pm", "90 minutes", 100));  
 *events*.put(2, new Event("UI Design", "Forth Valley College (Falkirk)", "25 July 2023", "12pm", "50 minutes", 250));  
 *events*.put(3, new Event("Malware Safety", "Cottrell Building", "2 August 2055", "4pm", "1 minute", 75));  
 }  
  
 */\*\*  
 \* Get a user from the User Details API  
 \** ***@param*** *uid the unique id of the user  
 \** ***@return*** *A User object if the uid corresponds to a user, null otherwise.  
 \*/* public User getUser(String uid){  
 try {  
 URL url = new URL(*USER\_DETAILS\_API* + "/user/" + uid);  
 HttpURLConnection connection = (HttpURLConnection) url.openConnection();  
  
 BufferedReader reader = new BufferedReader(new InputStreamReader(connection.getInputStream()));  
 String inputLine;  
 StringBuffer content = new StringBuffer();  
  
 while ((inputLine = reader.readLine()) != null) {  
 content.append(inputLine);  
 }  
  
 reader.close();  
 connection.disconnect();  
  
 JsonObject json1 = new JsonParser().parse(content.toString()).getAsJsonObject();  
 User user = new Gson().fromJson(json1.get("user"), User.class);  
  
 return user;  
 } catch (MalformedURLException e) {  
 e.printStackTrace();  
 } catch (IOException e) {  
 System.*out*.println("No user exists with given id (" + uid + ")");  
 }  
  
 return null;  
 }  
  
 */\*\*  
 \* Validates all users that are attending this event  
 \*/* public void validateAttendees(){  
 for(Event event : *events*.values()){  
 for(String user : event.getAttendees()){  
 if(!(getUser(user) instanceof User))  
 event.removeAttendee(user);  
 }  
 }  
 }  
  
 */\*\*  
 \* Get all attending users for an event  
 \* As a side effect, also validates all users going to an event before returning.  
 \** ***@param*** *eventId  
 \** ***@return*** *\*/* public List<User> getEventsAttendees(int eventId) {  
 Event event = *events*.get(eventId);  
  
 if (event == null) {  
 return null;  
 }  
  
 ArrayList<User> users = new ArrayList<>();  
 for (String userId : event.getAttendees()) {  
 User user = getUser(userId);  
  
 if (user == null) {  
 event.removeAttendee(userId);  
 } else {  
 users.add(user);  
 }  
 }  
  
 return users;  
 }  
  
 */\*\*  
 \* Get all currently registered Events  
 \** ***@return*** *An ArrayList of all Events  
 \*/* @Override  
 public ArrayList<Event> getAllEvents(){  
 validateAttendees();  
 return new ArrayList(*events*.values());  
 }  
  
 */\*\*  
 \* Get all currently registered Events  
 \** ***@return*** *An ArrayList of all Events  
 \*/* @Override  
 public ArrayList<Event> getAllEventsSanitized(){  
 validateAttendees();  
 ArrayList<Event> safeList = new ArrayList<>();  
  
 for(Event e : *events*.values()){  
 safeList.add(e.sanitise());  
 }  
  
 return safeList;  
 }  
  
 */\*\*  
 \* Get the amount of attendees going to an event  
 \** ***@param*** *eventID ID of an event to get attendee count for  
 \** ***@return*** *An int representing the amount of people going to the event  
 \*/* @Override  
 public int getEventAttendeesCount(int eventID) {  
 return *events*.get(eventID).getAttendeeCount();  
 }  
  
 */\*\*  
 \* Register a user as attending an event  
 \** ***@param*** *eventParam Event to register the User for  
 \** ***@param*** *user User that is going to the event  
 \*/* @Override  
 public boolean registerUser(Event eventParam, User user) {  
 Event event = *events*.get(eventParam.getEventId());  
 if(event != null) {  
 return event.addAttendee(user);  
 }  
  
 return false;  
 }  
  
 */\*\*  
 \* Unregister a user from an event  
 \** ***@param*** *eventParam Event to unregister the User from  
 \** ***@param*** *user User that is no longer going to the Event  
 \*/* @Override  
 public boolean unregisterUser(Event eventParam, User user) {  
 Event event = *events*.get(eventParam.getEventId());  
  
 if(event != null) {  
 return event.removeAttendee(user);  
 }  
  
 return false;  
 }  
  
 */\*\*  
 \* Store a new event in the system  
 \** ***@param*** *event Event to register  
 \** ***@return*** *The unique EventId of the new event  
 \*/* @Override  
 public int createEvent(Event event) {  
 Event newEvent = new Event(event.getDescription(), event.getLocation(), event.getDate(), event.getTime(), event.getDuration(), event.getMaxCapacity());  
 *events*.put(newEvent.getEventId(), newEvent);  
 return newEvent.getEventId();  
 }  
  
 */\*\*  
 \* Get all Events a user is attending  
 \** ***@param*** *userId Unique UserID to get events for  
 \** ***@return*** *An ArrayList of Events the user is going to.  
 \*/* @Override  
 public List<Event> getUsersEvents(String userId) {  
 List<Event> attendingEvents = new ArrayList<>();  
  
 for(Event event : *events*.values()){  
 if(event.hasAttendee(userId)){  
 attendingEvents.add(event);  
 }  
 }  
  
 System.*out*.println(attendingEvents);  
  
 return attendingEvents.size() > 0 ? attendingEvents : null;  
 }  
  
 @Override  
 public boolean deleteEvent(int eventId) {  
 return *events*.remove(eventId) != null;  
 }  
  
 */\*\*  
 \* Checks whether a user with the given uid exists in the User Details API  
 \*  
 \** ***@param*** *uid [String] uid to validate  
 \** ***@return*** *True if a user with the given id exists  
 \*/* @Override  
 public boolean authenticate(String uid){  
 return getUser(uid) != null;  
 }  
  
 */\*\*  
 \* Gets an event with the given ID, or null if no event exists.  
 \** ***@param*** *id  
 \** ***@return*** *\*/* @Override  
 public Event getEvent(int id) {  
 return *events*.get(id);  
 }  
}

## Controller Class

@RestController  
@CrossOrigin  
public class Controller {  
 private final EventService service;  
  
 public Controller(EventService eventService){  
 service = eventService;  
 }  
  
 */\*\*  
 \* UNUSED, was originally going to make a login screen but couldnt be bothered. too much diss work, too many assignments, not enough time.  
 \*  
 \* Send a request to Login  
 \** ***@param*** *user  
 \** ***@param*** *response  
 \*/* @Deprecated  
 @PostMapping("/login")  
 public void login(@RequestBody LoginFormSubmission user, HttpServletResponse response){  
 if(service.authenticate(user.userid())){  
 response.setStatus(HttpServletResponse.*SC\_OK*);  
 response.setHeader("Access-Control-Allow-Origin", "\*");  
  
 return;  
 }  
  
 response.setStatus(HttpServletResponse.*SC\_UNAUTHORIZED*);  
 }  
  
 */\*\*  
 \* Request all stored events  
 \** ***@param*** *userid [Optional] A userId, if supplied will only get events this user is registered to  
 \** ***@param*** *response  
 \** ***@return*** *A list of all events, or events a given user is registered to  
 \*/* @GetMapping("/events")  
 public List<Event> getEvents(@RequestParam(required = false) String userid, HttpServletResponse response){  
 response.setHeader("Access-Control-Allow-Origin", "\*");  
 return userid != null ? service.getUsersEvents(userid) : service.getAllEventsSanitized();  
 }  
  
  
 */\*\*  
 \* Get an event with a given ID  
 \** ***@param*** *id  
 \** ***@param*** *response  
 \** ***@return*** *an Event with the associated ID  
 \** ***@return*** *a Error 404 (Not Found) if no Event with the given ID is stored  
 \*/* @GetMapping("/events/{id}")  
 public Event getEvent(@PathVariable int id, HttpServletResponse response){  
 response.setHeader("Access-Control-Allow-Origin", "\*");  
 Event event = service.getEvent(id);  
  
 if(event == null){  
 System.*out*.println("No event with that id found");  
 response.setStatus(HttpServletResponse.*SC\_NOT\_FOUND*);  
 return null;  
 }  
  
 response.setStatus(HttpServletResponse.*SC\_OK*);  
 return event;  
 }  
  
 */\*\*  
 \* Register a user for an event  
 \** ***@param*** *data [RegisterSubmission] Data passed in the POST body  
 \** ***@param*** *response  
 \** ***@return*** *Error 400 (Bad Request) if no data or invalid format was supplied  
 \** ***@return*** *Error 409 (Conflict) if the user is already registered for the event  
 \** ***@return*** *200 (OK) if the user is registered successfully.  
 \*/* @PutMapping("/events/register")  
 public void registerForEvent(@RequestBody RegisterSubmission data, HttpServletResponse response){  
 if(data == null){  
 System.*out*.println("Malformed User or Event data");  
 response.setStatus(HttpServletResponse.*SC\_BAD\_REQUEST*);  
 return;  
 }  
  
 if(service.registerUser(data.event(), data.user())) {  
 System.*out*.println("Registered user to event");  
 response.setStatus(HttpServletResponse.*SC\_OK*);  
 return;  
 }  
  
 System.*out*.println("User is already registered to that event");  
 response.setStatus(HttpServletResponse.*SC\_CONFLICT*);  
 }  
  
 */\*\*  
 \* Unregister a user  
 \*  
 \** ***@param*** *data  
 \** ***@param*** *response  
 \** ***@return*** *Error 400 (Bad Request) if the data supplied is invalid or null  
 \** ***@return*** *Error 409 (Conflict) if the user is not registered to the event  
 \** ***@return*** *200 (OK) if the user is successfully unregistered  
 \*/* @PutMapping("/events/unregister")  
 public void unregisterForEvent(@RequestBody RegisterSubmission data, HttpServletResponse response){  
 if(data == null){  
 System.*out*.println("Malformed User or Event data");  
 response.setStatus(HttpServletResponse.*SC\_BAD\_REQUEST*);  
 return;  
 }  
  
 if(service.unregisterUser(data.event(), data.user())){  
 response.setStatus(HttpServletResponse.*SC\_OK*);  
 System.*out*.println("Unregistered user");  
 return;  
 }  
  
 System.*out*.println("User is not registered for event");  
 response.setStatus(HttpServletResponse.*SC\_CONFLICT*);  
 }  
  
 private boolean authenticateAdminPassword(String input){  
 String valid = "Basic " + Base64.*getEncoder*().encodeToString("admin:password1".getBytes(StandardCharsets.*UTF\_8*));  
 return valid.equals(input);  
 }  
  
 */\*\*  
 \* Check if the user supplies an admin password  
 \* this protocol is quite possible the most insecure code i've ever written  
 \** ***@param*** *base64password  
 \** ***@param*** *response  
 \** ***@return*** *\*/* @GetMapping("admin/auth")  
 public boolean authenticateAdmin(@RequestHeader("Authorization") String base64password, HttpServletResponse response){  
 response.setHeader("Access-Control-Allow-Origin", "\*");  
  
 if(authenticateAdminPassword(base64password)){  
 System.*out*.println("Authenticated administrator user");  
 response.setStatus(HttpServletResponse.*SC\_OK*);  
 return true;  
 }  
  
 System.*out*.println("Admin password is incorrect");  
 response.setStatus(HttpServletResponse.*SC\_UNAUTHORIZED*);  
 return false;  
 }  
  
 */\*\*  
 \* Create a new event  
 \* Requires administrator password to be passed in Credentials header  
 \** ***@param*** *event  
 \** ***@param*** *response  
 \** ***@return*** *Error 401 (Unauthorized) if no or invalid Credential header is supplied  
 \** ***@return*** *Error 400 (Bad Request) if no or invalid Event data is supplied  
 \** ***@return*** *200 (OK) if the event is successfully created, with the new event path in the Location response header  
 \*/* @PostMapping("/admin/create-event")  
 public void createEvent(@RequestBody Event event, @RequestHeader("Authorization") String base64password, HttpServletResponse response){  
 if(!authenticateAdminPassword(base64password)){  
 System.*out*.println("no admin auth provided");  
 response.setStatus(HttpServletResponse.*SC\_UNAUTHORIZED*);  
 return;  
 }  
  
 if(event == null){  
 System.*out*.println("Malformed event format");  
 response.setStatus(HttpServletResponse.*SC\_BAD\_REQUEST*);  
 return;  
 }  
  
 System.*out*.println("Created new event");  
 int newEventId = service.createEvent(event);  
  
 response.setHeader("Access-Control-Allow-Origin", "\*");  
 response.setHeader("Location", "/events/" + newEventId);  
 response.setStatus(HttpServletResponse.*SC\_OK*);  
 }  
  
 */\*\*  
 \* Get all attendees for an event, requires administrator password in Credentials header  
 \** ***@param*** *eventid  
 \** ***@param*** *response  
 \** ***@return*** *Error 401 (Unauthorized) if invalid or no credentials supplied  
 \** ***@return*** *Error 404 (Not Found) if no event could be found for the given ID  
 \** ***@return*** *200 (OK) if the attendees list is returned successfully  
 \*/* @GetMapping("/admin/get-attendees")  
 public List<User> getAttendees(@RequestParam int eventid, @RequestHeader("Authorization") String base64password, HttpServletResponse response){  
 System.*out*.println(base64password);  
 if(!authenticateAdminPassword(base64password)){  
 System.*out*.println("No admin auth provided");  
 response.setStatus(HttpServletResponse.*SC\_UNAUTHORIZED*);  
 return null;  
 }  
  
 List<User> attendees = service.getEventsAttendees(eventid);  
 if(attendees == null){  
 System.*out*.println("No event with specified ID found");  
 response.setStatus(HttpServletResponse.*SC\_NOT\_FOUND*);  
 return null;  
 }  
  
 response.setStatus(HttpServletResponse.*SC\_OK*);  
 return attendees;  
 }  
  
 */\*\*  
 \* Deletes an event with a given EventId  
 \** ***@param*** *eventid EventID to delete from the system  
 \** ***@param*** *response  
 \*  
 \** ***@returns*** *Error 400 (Bad Request) if the event ID is not given, or malformed.  
 \** ***@returns*** *Error 404 (Not Found) if the ID is valid but no event exists associated with it  
 \** ***@returns*** *204 (No Content) if the event was successfully deleted from the system.  
 \*/* @DeleteMapping("/admin/delete-event")  
 public void deleteEvent(@RequestParam int eventid, @RequestHeader("Authorization") String base64password, HttpServletResponse response){  
 if(!authenticateAdminPassword(base64password)){  
 System.*out*.println("No admin authentication provided");  
 response.setStatus(HttpServletResponse.*SC\_UNAUTHORIZED*);  
 return;  
 }  
  
 if(service.deleteEvent(eventid)){  
 System.*out*.println("Deleted event");  
 response.setStatus(HttpServletResponse.*SC\_NO\_CONTENT*);  
 }else{  
 System.*out*.println("No event with specified ID");  
 response.setStatus(HttpServletResponse.*SC\_NOT\_FOUND*);  
 }  
 }  
}

# JavaScript Code

## Admin.js

function createNewEvent(){  
 let description = ***document***.getElementById('event-description').value;  
 let location = ***document***.getElementById('event-location').value;  
 let date = ***document***.getElementById('event-date').value;  
 let time = ***document***.getElementById('event-time').value;  
 let duration = ***document***.getElementById('event-duration').value;  
 let capacity = ***document***.getElementById('event-capacity').value;  
  
 if(description == "" || description == null){  
 alert("Please enter a valid description");  
 }else if (location == "" || location == null){  
 alert("Please enter a valid location");  
 }else if (date == "" || location == null){  
 alert("Please enter a valid date");  
 }else if (time == "" || location == null){  
 alert("Please enter a valid time");  
 }else if (duration == "" || location == null){  
 alert("Please enter a valid duration");  
 }else if (capacity == "" || location == null){  
 alert("Please enter a valid capacity");  
 }  
  
 let eventObj = new ***Object***()  
 eventObj.description = description;  
 eventObj.location = location;  
 eventObj.date = date;  
 eventObj.time = time;  
 eventObj.duration = duration;  
 eventObj.maxCapacity = capacity;  
  
 ***console***.log(eventObj);  
 fetch('http://localhost:8080/admin/create-event', {  
 method: 'POST',  
 headers: {  
 'Content-Type': 'application/json',  
 'Authorization': 'Basic ' + btoa('admin:' + ***sessionStorage***.getItem("AdminPass"))  
 },  
 body: ***JSON***.stringify(eventObj)  
 }).  
 then(response => response.json()).  
 catch(error => ***console***.log(error));  
}  
  
function switchContent(divToShow){  
 let eventsTableDiv = ***document***.getElementById('viewEvents');  
 let createEventDiv = ***document***.getElementById('createEvent');  
 divToShow = ***document***.getElementById(divToShow);  
  
 if(divToShow == eventsTableDiv){  
 eventsTableDiv.style.display = 'inline';  
 createEventDiv.style.display = 'none';  
 getEvents();  
 }else{  
 eventsTableDiv.style.display = 'none';  
 createEventDiv.style.display = 'inline';  
 }  
}  
  
function getEvents(){  
 fetch('http://localhost:8080/events', {  
 headers: {  
 'Authorization': 'Basic ' + btoa('admin:' + ***sessionStorage***.getItem("AdminPass"))  
 }  
 }).  
 then(response => response.json()).  
 then(data => populateAdminEvents(data)).  
 catch(error => ***console***.log(error));  
}  
  
function clearTable(eventsTable){  
 const rowCount = eventsTable.rows.length;  
 for (let i = rowCount - 1; i > 0; i--) {  
 eventsTable.deleteRow(i);  
 }  
}  
  
function populateAdminEvents(json){  
 const eventsTable = ***document***.getElementById("eventsTable");  
  
 clearTable(eventsTable);  
  
 for(let i = 0; i < ***Object***.keys(json).length; i++){  
 let newRow = eventsTable.insertRow(i+1);  
  
 newRow.insertCell(0).innerHTML = json[i].eventId;  
 newRow.insertCell(1).innerHTML = json[i].description;  
 let button = newRow.insertCell(2);  
  
 button.innerHTML = "<td>" +  
 "<button style='margin: 5px' onclick='deleteAnEvent(\"" + json[i].eventId + "\")'>Delete</button>" +  
 "<button style='margin: 5px' onclick='viewAttendees(\"" + json[i].eventId + "\")'>View Attendees</button>" +  
 "</td>"  
 }  
}  
  
function viewAttendees(eventId){  
 let attendeesTable = ***document***.getElementById('attendeesTable');  
  
 const rowCount = attendeesTable.rows.length;  
 for (let i = rowCount - 1; i > 0; i--) {  
 attendeesTable.deleteRow(i);  
 }  
  
 fetch('http://localhost:8080/admin/get-attendees?eventid=' + eventId, {  
 headers: {'Authorization': 'Basic ' + btoa('admin:' + ***sessionStorage***.getItem("AdminPass"))}  
 }).  
 then(response => response.json()).then(function(data){  
 ***console***.log(data[0]);  
  
  
 for(let i = 0; i < ***Object***.keys(data).length; i++){  
 let newRow = attendeesTable.insertRow(i+1);  
  
 let userId = newRow.insertCell(0);  
 let name = newRow.insertCell(1);  
 let button = newRow.insertCell(2);  
  
 userId.innerHTML = data[i].uid;  
 name.innerHTML = data[i].name;  
  
 button.innerHTML = "<button ='margin: 5px' onclick='unregisterUser(\"" + data[i].uid + "\", " + eventId + ")'>Force Remove</button>"  
 }  
 })  
}  
  
function unregisterUser(userId, eventId){  
 let obj = new ***Object***();  
 obj.userId = userId;  
 obj.eventId = eventId;  
  
 fetch( 'http://localhost:8080/events/' + "unregister", {  
 method: 'PUT',  
 headers: {  
 'Content-Type': 'application/json'  
 },  
 body: ***JSON***.stringify(obj)  
 }).  
 then(function(response){  
 if(response.status == 409){  
 alert("You are not registered to this event");  
 }else{  
 alert("You have unregistered user " + userId);  
 }  
  
 viewAttendees(eventId);  
 }).  
 catch(error => ***console***.log(error));  
}  
  
function deleteAnEvent(eventId){  
 fetch('http://localhost:8080/admin/delete-event?eventid=' + eventId, {  
 method: 'DELETE',  
 headers: {  
 'Content-Type': 'application/json',  
 'Authorization': 'Basic ' + btoa('admin:' + ***sessionStorage***.getItem("AdminPass"))  
 },  
 }).then(function(response){  
 switchContent('viewEvents')  
 });  
}  
  
  
function verifyAdminPassword(){  
 let password = prompt("Please enter admin password")  
  
 fetch('http://localhost:8080/admin/auth', {  
 method: 'GET',  
 headers: {  
 'Authorization': 'Basic '+btoa('admin:' + password)   
 }  
 })  
 .then(response => response.json()  
 .then(function (serversResponse){  
 if(serversResponse){  
 ***sessionStorage***.AdminPass = password  
 }else{  
 ***location***.reload()  
 }  
 }))  
}  
verifyAdminPassword()

## Events.js

function getAllEvents(){  
 fetch('http://localhost:8080/events').  
 then(response => response.json()).  
 then(data => populateTable(data)).  
 catch(error => ***console***.log(error));  
}  
  
function getUsersEvents(){  
 let userid = ***document***.getElementById('userIdSearch').value;  
 if(userid == null || userid == ""){  
 alert("You must enter your user id before searching");  
 return;  
 }  
  
 fetch('http://localhost:8080/events?userid=' + userid).  
 then(function(response) {  
 if(response == null){  
 return  
 }  
 response.json().then(data => populateTable(data));  
 }).  
 catch(error => ***console***.log(error));  
}  
  
function clearTable(eventsTable){  
 const rowCount = eventsTable.rows.length;  
 for (let i = rowCount - 1; i > 0; i--) {  
 eventsTable.deleteRow(i);  
 }  
}  
  
function populateTable(json){  
 const eventsTable = ***document***.getElementById("eventsTable");  
  
 clearTable(eventsTable);  
  
 for(let i = 0; i < ***Object***.keys(json).length; i++){  
 let newRow = eventsTable.insertRow(i+1);  
  
 let eventDescription = newRow.insertCell(0);  
 let button = newRow.insertCell(1);  
  
 eventDescription.innerHTML = json[i].description;  
 button.innerHTML = "<td><button onclick='viewEvent(" + json[i].eventId + ")'>View More</button></td>"  
 }  
  
 eventsTable.insertRow(0);  
}  
  
function viewEvent(id){  
 ***window***.location.href = "event\_view.html?id=" + id;  
}  
  
***document***.addEventListener("DOMContentLoaded", function(){  
 const urlParameters = new ***URLSearchParams***(***window***.location.search);  
 let userId = urlParameters.get('user');  
  
 if(userId != null){  
 getUsersEvents(userId);  
 }else{  
 getAllEvents();  
 }  
})

## Event\_view.js

let ***eventId***;  
let ***url*** = 'http://localhost:8080/events/'  
  
function getEventDetails(id){  
 ***eventId*** = id;  
 fetch( ***url*** + id).  
 then(response => response.json()).  
 then(data => fillEventDetails(data)).  
 catch(error => ***console***.log(error));  
}  
  
function getEventAttendees(id){  
 fetch(***url*** + id).  
 then(response => response.json()).  
 then(data => fillEventDetails(data)).  
 catch(error => ***console***.log(error));  
}  
  
function fillEventDetails(json){  
 ***document***.getElementById('event\_title').innerHTML = json.description;  
 ***document***.getElementById('event\_description').innerHTML = "Description: " + json.description;  
 ***document***.getElementById('event\_location').innerHTML = "Location: " + json.location;  
 ***document***.getElementById('event\_date').innerHTML = "Date: " + json.date;  
 ***document***.getElementById('event\_time').innerHTML = "Time: " + json.time;  
 ***document***.getElementById('event\_duration').innerHTML = "Duration: " + json.duration;  
 ***document***.getElementById('event\_capacity').innerHTML = "Capacity: " + json.maxCapacity;  
 ***document***.getElementById('event\_register').innerHTML = "<button type='button' class='btn btn-outline-primary' onclick='registerForEvent()'>Register</button><button type='button' class='btn btn-outline-primary' onclick='unregisterForEvent()'>Unregister</button>"  
  
}  
  
function registerForEvent(){  
 let user\_id = ***document***.getElementById('user-id').value;  
  
 if(user\_id == null || user\_id == ""){  
 alert("You must enter your user ID");  
 return  
 }  
  
 let obj = new ***Object***();  
 obj.userId = user\_id;  
 obj.eventId = ***eventId***;  
  
 fetch( ***url*** + "register", {  
 method: 'PUT',  
 headers: {  
 'Content-Type': 'application/json'  
 },  
 body: ***JSON***.stringify(obj)  
 }).  
 then(function(response){  
 if(response.status == 409){  
 alert("You are already registered for this event");  
 }else{  
 alert("You have registered for this event")  
 }  
 }).  
 catch(error => prompt);  
}  
  
function unregisterForEvent(){  
 let user\_id = ***document***.getElementById('user-id').value;  
  
 if(user\_id == null || user\_id == ""){  
 alert("You must enter your user ID");  
 return  
 }  
  
 let obj = new ***Object***();  
 obj.userId = user\_id;  
 obj.eventId = ***eventId***;  
  
 fetch( ***url*** + "unregister", {  
 method: 'PUT',  
 headers: {  
 'Content-Type': 'application/json'  
 },  
 body: ***JSON***.stringify(obj)  
 }).  
 then(function(response){  
 if(response.status == 409){  
 alert("You are not registered to this event");  
 }else{  
 alert("You have unregistered");  
 }  
 }).  
 catch(error => ***console***.log(error));  
}  
  
***document***.addEventListener("DOMContentLoaded", function(){  
 const querystr = ***window***.location.search;  
 const urlParameters = new ***URLSearchParams***(querystr);  
  
 getEventDetails(urlParameters.get('id'));  
})