University of Stirling

Computing Science

Event registration application

CSCU9YW Web Services Assignment

Student Number: 2944806

Stirling, April 2022

Table of Contents

Introduction

This report describes a web-based event registration application. It consists of an API running on a server and a client application running in a web browser. The API is implemented in Java in Spring Boot. The client application is implemented in TypeScript and combines a public attendee view with a secured admin view.

Attendee view allows visitors to register a new user profile from an external user profile service. These users can then register for events and cancel their registrations. They are also able to view their interests and the events they are currently registered to.

Admin view has a lot more functionalities. It allows administrators to manage the events – create new ones, edit, and delete existing ones. Moreover, they can view the attendees of the events, including their interests. Besides that, they are also able to validate the attendees of an event. This action requests all the event's registered users from the external user profile service. If the user no longer exists in the external service, the user is unregistered from all the events, and their profile is deleted from the database. If the user's interests are changed, they are updated in the application's database. This functionality is implemented on the server-side. The client application communicates solely with the API.

Both the API and the client applications were deployed and are available on the Internet. Access details are provided in the relevant chapters 1.5 and 2.3.

1 Server

1.1 Data structure

The data structure is straightforward; it consists of only a few tables. The service uses an in-memory H2 SQL database at the moment, but it should be reasonably easy to store the data persistently or change the DBMS altogether just by using a different database driver.

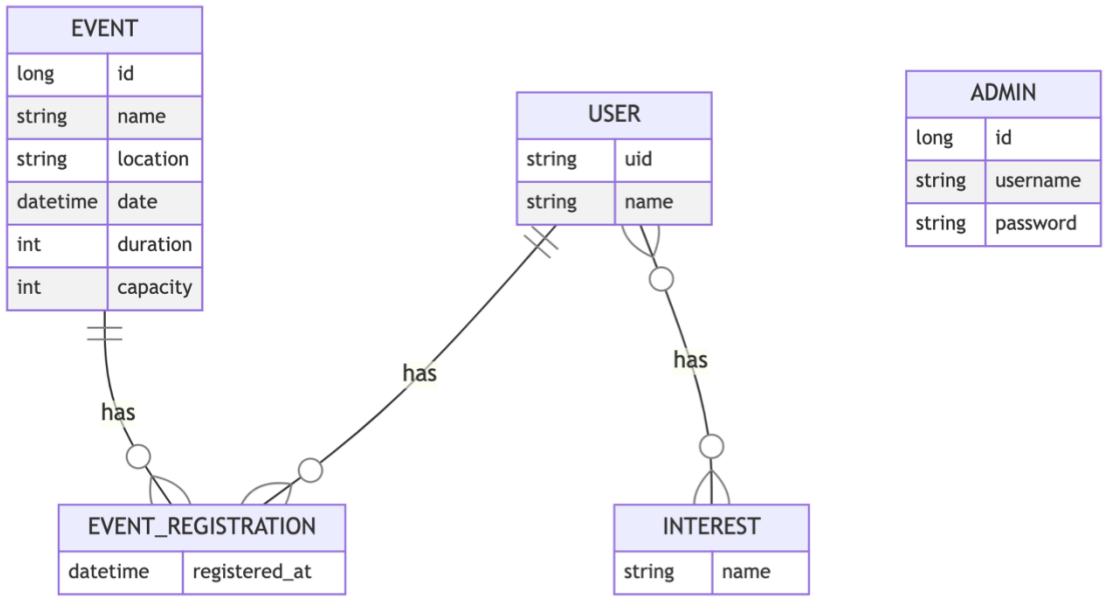


Figure 1: ER diagram of the service

1.2 REST architecture

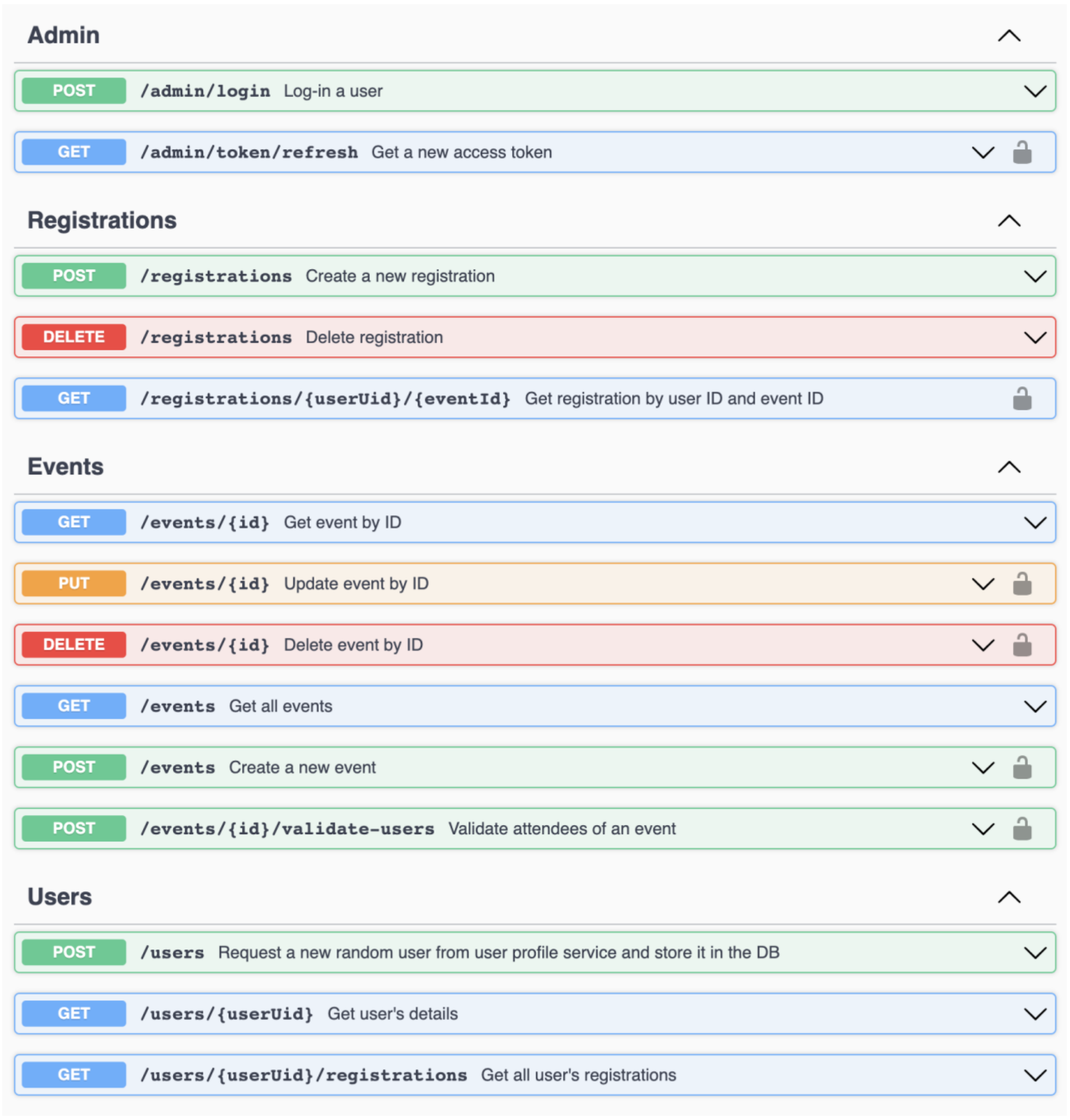
I believe that the service follows most of the REST design principles. All data are accessible via URLs and represented in the JSON format. The service is stateless because it doesn't hold any session data about users. The only principle the service doesn't follow is the HATEOAS. All the details about the API architecture are included in the API documentation that can be found in *Appendix A: API design*. A link to a slightly more interactive web representation (Swagger UI) of the same OpenAPI 3 documentation can be found in chapter 1.5.

1.3 Security

About half of the endpoints require a valid Bearer token in the Authorization header. Those are the endpoints that are available only for users with administrative privileges. The whole token flow starts by logging in using the `/admin/login` endpoint. The admin is asked to fill in their username and password, and if the credentials are correct, the server responds with a pair of JWT tokens – the first is called access token and the latter being called refresh token. Both are signed by a server secret. The access token is valid for only one minute in the case of my application (its validity would probably be longer in a production environment). The refresh token is valid for 2 hours. The access token must be used as a Bearer token in the `Authorization` header for all protected endpoints. If the access token is expired, the server responds with an error with the HTTP response code 401. The client app is expected to request a new access token by calling the `/admin/token/refresh` endpoint while using the refresh token in the `Authorization` header. The API replies with a new access token if the refresh token is valid. If the refresh token is expired as well, an error 401 response is sent, and the client app is expected to log out the current user and ask them to log in using the username-password combination again to acquire a fresh access/refresh token pair.

Some other endpoints – e.g., the `/events` endpoints provide data for unauthorized and authorized users, but there is a difference in the response body. The field with information about registered users for a particular event is available only for authorized admin users with a valid access token. It is ensured by two different JSON view classes. The rest of the endpoints – those are the endpoints used in the attendee client – are not secured at all. If you know the UID of a user, you will be able to access their user data and register and unregister them for events. The only validation is a check whether a value of the `X-User-Uid` header is equal to the `userUid` query parameter in case of the `/users` endpoints or equal to the value of the `userUid` field in case of `DELETE` and `POST` requests of the endpoint `/registrations`. The API documentation clearly indicates which endpoints require a valid access token in the `Authorization` header by a padlock icon and those that require an `X-User-Uid` header.

Credentials for the default administrator user can be changed in the application.properties file included in the ZIP archive.



Admin

POST `/admin/login` Log-in a user

GET `/admin/token/refresh` Get a new access token 🔒

Registrations

POST `/registrations` Create a new registration

DELETE `/registrations` Delete registration

GET `/registrations/{userUid}/{eventId}` Get registration by user ID and event ID 🔒

Events

GET `/events/{id}` Get event by ID

PUT `/events/{id}` Update event by ID 🔒

DELETE `/events/{id}` Delete event by ID 🔒

GET `/events` Get all events

POST `/events` Create a new event 🔒

POST `/events/{id}/validate-users` Validate attendees of an event 🔒

Users

POST `/users` Request a new random user from user profile service and store it in the DB

GET `/users/{userUid}` Get user's details

GET `/users/{userUid}/registrations` Get all user's registrations

Figure 2: All available REST endpoints. Padlocks indicate which routes require a valid access token.

1.4 Performance and scalability

In the current state, the service uses an in-memory database (H2). The database would need to be replaced with a different DB running outside of the application to enable the app's scalability. It would allow the application itself to be run across multiple server instances simultaneously, and all instances would be able to access the same data. Of course, the data would have to be stored persistently in the production environment. Naturally, the DB could act as a performance bottleneck too, but there are a few techniques for dealing with such limitations of centralized databases. It is out of the scope of this module, though. Using stateless JWT tokens for authorization also helps with scalability, because the server doesn't hold any session data.

1.5 Continuous deployment

The service was deployed to Google Cloud's Cloud Run platform (https://cloud.google.com/run). The setup was easier than I initially expected and was comparable to deploying Node.js applications there. A new revision is automatically deployed when a new code is pushed to the `main` branch of the project's Git repository. The build process is provided by Google's Cloud Build service. This is the URL of the deployed API: https://event-registration-api-mrl4z6unya-ey.a.run.app/api/v1. Please note that a cold start may take up to 30 seconds, and you will be able to visit only `/events` endpoint in your browser. You can also visit the API documentation there. It is available at https://event-registration-api-mrl4z6unya-ey.a.run.app/api/v1/swagger-ui/index.html.
The deployed service requires the use of HTTPS; all requests requesting the unsecured HTTP protocol are redirected to HTTPS. Because an in-memory database is currently used, all data are lost after 15 minutes of inactivity. I find it quite beneficial for demonstrational purposes like this because the application always starts in a clean state with only a predefined set of events. The deployed application uses its own set of environment variables because the application.properties file is not included in the Git repository. For demonstration purposes, the default admin credentials for the deployed service are set to be the same as the ones that can be found in the application.properties file in the attached ZIP archive (variables `admin.username` and `admin.password`).

2 Client

2.1 Technologies

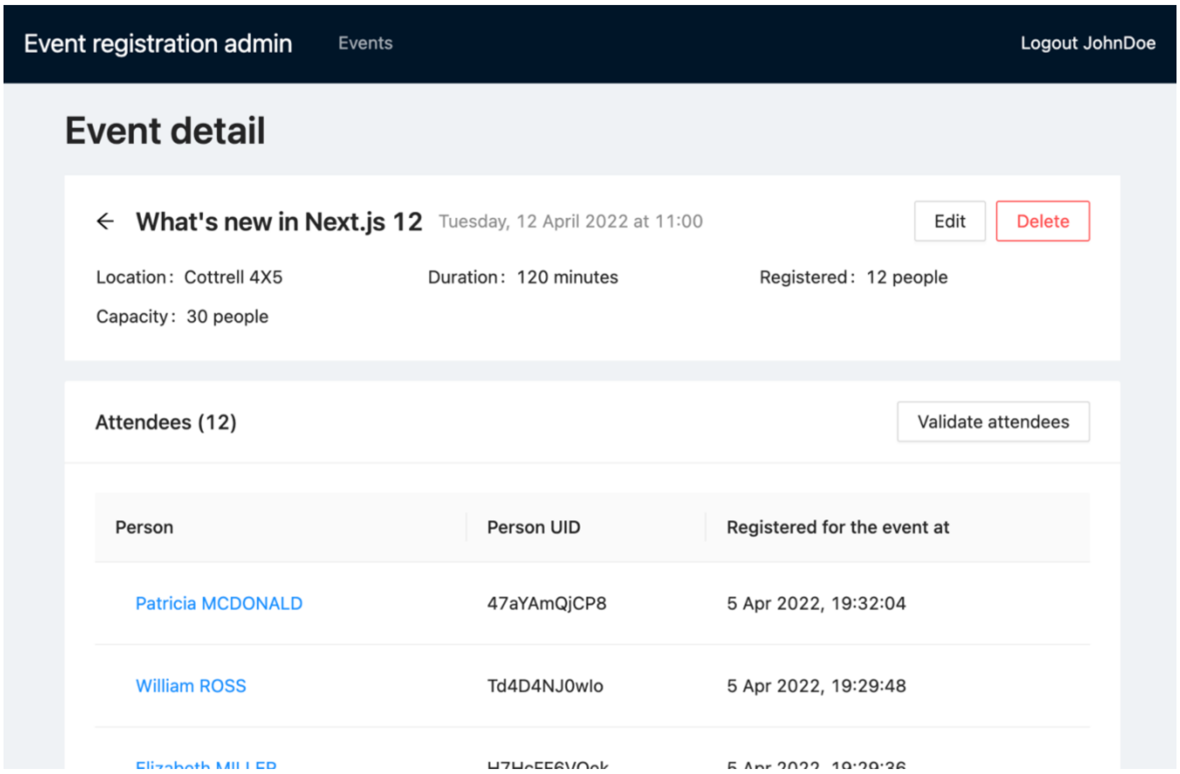
I decided to code the frontend client application in TypeScript, specifically using React library. TypeScript is a superset of JavaScript that adds static typing. It gave me the ability to use the same types on the client as on the server. The TypeScript code must be transpiled to JavaScript before it can be run in the browser, but the tooling around it is pretty advanced nowadays, the transpilaion step is executed automatically in the background. Therefore, the developer experience is excellent.

2.2 Security and JWT token flow

Each API request and response is intercepted using so-called Axios Interceptors. Axios is an HTTP client I decided to use instead of the native `fetch()` API, mainly for this intercepting ability. It allows the client app to add the `Authorization` header to each request requesting data from protected API routes and to intercept every response that has HTTP 401 status code, and the value of `cause` property in returned JSON is `TOKEN_EXPIRED`. In that case, a new access token is requested from `/admin/token/refresh` endpoint (this is the only request that uses the refresh token in the `Authorization` header. All other requests to protected routes use the access token). If the refresh token is valid, the server generates a new JWT access token and sends it back to the client application. The client application immediately sends another request to the originally requested resource, but this time with the new access token and the resource data is then sent back to the client. This process is automated and happens in the background. The end-user shouldn't notice that such an operation was performed. The only case when the user notices it is if the refresh token is expired as well (the server responds with HTTP 401 and cause `TOKEN_EXPIRED` to the request to the `/refresh` endpoint). Given this scenario, the user is logged out from the client application and is asked to log in again (which effectively generates a new refresh token).

2.3 Continuous deployment

The frontend application is deployed automatically after each push to the `main` branch of the Git repository. The provider was chosen to be Vercel (https://vercel.com/) mainly for its hassle-free integration with GitHub. The setup took less than 10 minutes in total. The application can be visited at https://cscu9yw-event-registration.vercel.app/. Please bear in mind that the cold start of the API server can take up to 30 seconds, as mentioned in 1.5. Thus, it may be necessary to refresh the client application if the API request times out. After the first start, you should not observe any other delays.


Event registration admin Events Logout JohnDoe

Event detail

What's new in Next.js 12 Tuesday, 12 April 2022 at 11:00 Edit Delete

Location: Cottrell 4X5 Duration: 120 minutes Registered: 12 people

Capacity: 30 people

Attendees (12) Validate attendees

Person	Person UID	Registered for the event at
Patricia MCDONALD	47aYAmQjCP8	5 Apr 2022, 19:32:04
William ROSS	Td4D4NJ0wlo	5 Apr 2022, 19:29:48
Elizabeth MILLER	H7HcFF6VQok	5 Apr 2022, 19:29:36

Figure 3: Admin view of the client application