Technical Documentation

This documentation will go over our designs, project architecture, the integration with Strava and our backend and frontend technologies.

A quick refresher

The application we've built was made using Svelte and Java Spring Boot. The application allows the people who sign up to compete in sports challenges, made by other people or themselves, in order to promote competition. The challenges could have a goal like "Run 15km every week". The challenge itself could then take place over a month, with weekly intervals for example. Whomever runs 15km in a week in this instance will have that period of the challenge marked as completed. There are challenges where users compete against each other, but also ones for teams. A user can also set up a personal challenge, which we call a streak. The user can manually log activities inside of the application, but can also connect their Strava account so their activities get pulled in automatically. Any manually logged activity will have to be approved by an admin of the application, in order to try to prevent cheating the system.

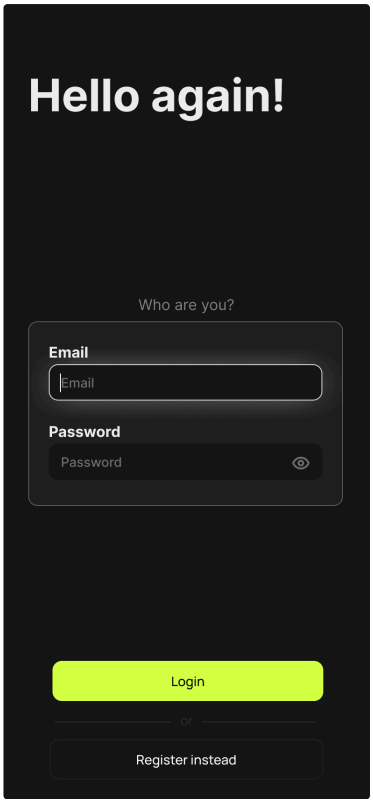
Designs and application flow

This page contains our Figma designs we based our actual frontend on. We didn't design every page since most of it was subject to change anyway.

Before we started designing, we first went looking for sport app designs online. We took some of the best looking ones and based our application on a mix of all of them.

Authentication

Login

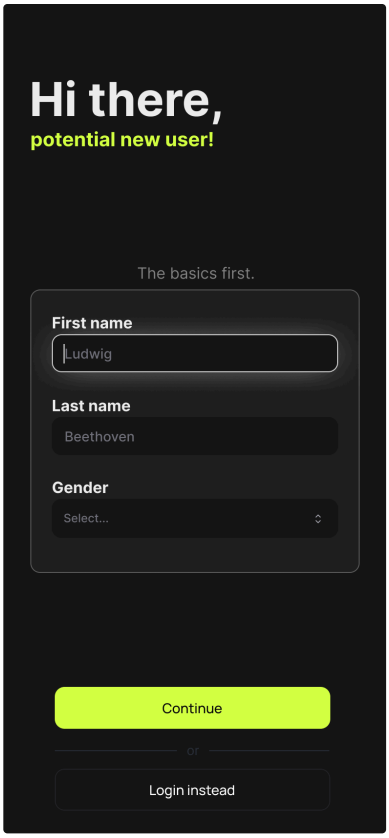


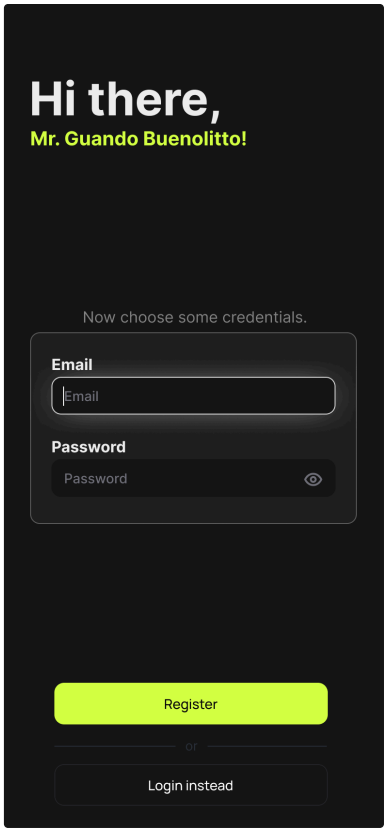
The login screen welcomes the user at the very top in a friendly manner.

In the center of the screen is a login form where the user can enter their credentials. The *password* field has an eye icon used to toggle between a text or hidden password.

On the bottom of the page are the navigation buttons. One for logging in and one for navigating to the register page instead.

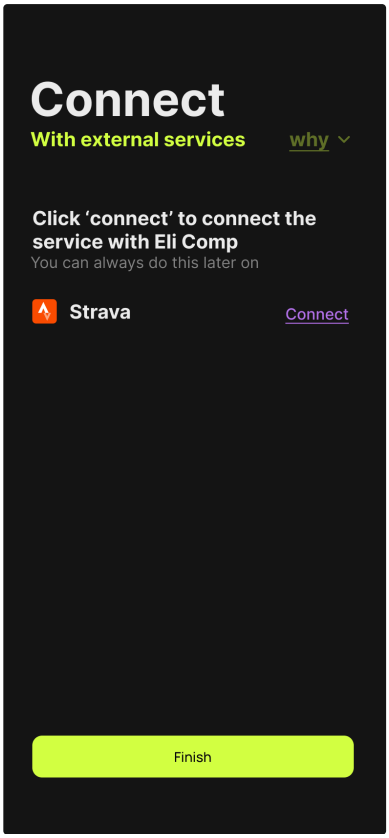
Register





Instead of being a single large page, registering is a multi-step undertaking consisting of 3 pages.

It starts of with asking for some personal information like a first name, last name and gender. After clicking on continue, the user will be taken to the second step. Much like on the login page there is a possibility to navigate away from this page and to login instead. On the second page the user is greeted and then prompted to choose an email and a password. After clicking on *register* an account will be made for them and they will be taken to the last page of the register flow.



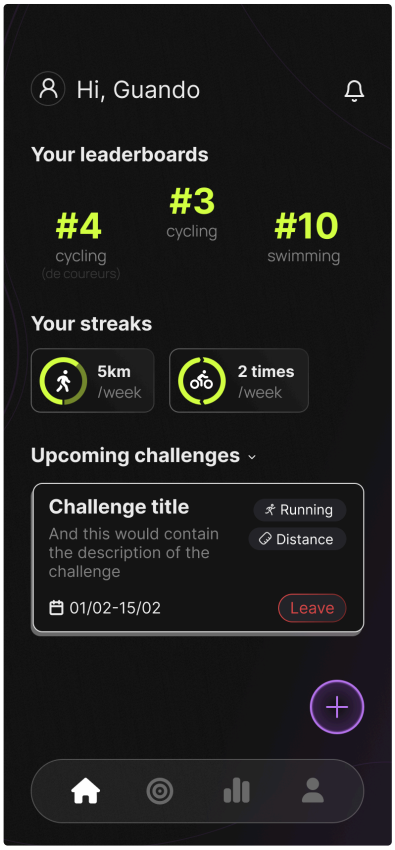
On this page the user can choose to link their Elicomp to a Strava account. They can click on why in order to learn why they should do this. It's also shown to the user that this is optional.

Near the center Strava is displayed as an option for pulling in activities. It is accompanied by a *connect* button that will start the OAuth process.

After clicking on *Finish* the user will be taken to their homepage.

Application

Homepage



The homepage consists of 5 sections.

At the top of the page the user is greeted. They'll be able to see their profile picture and see whatever notifications they have.

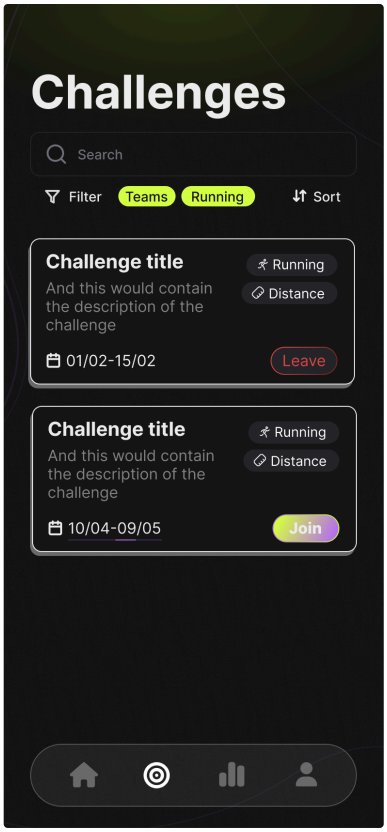
Below that is the leaderboard section. Here they'll be able to see their leaderboard standings. These could be personal ones or ones for some of the teams they've joined. These should be customizable by the user. They should be able to go to their profile settings and choose what standings they want to display on their homepage.

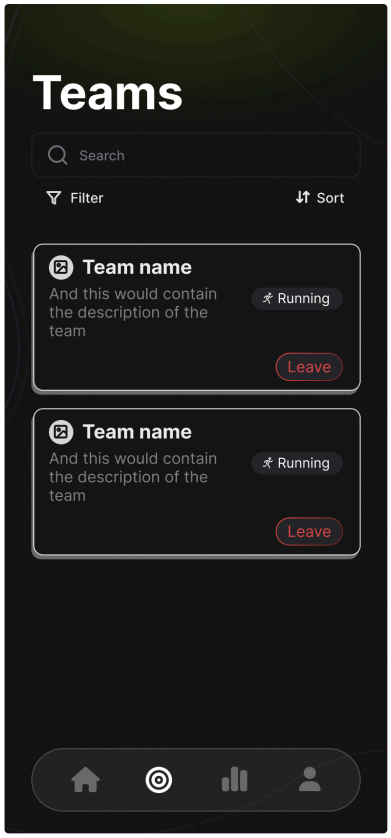
Next up is the streak section where the user's personal challenges, we name them streaks, are displayed. The user can see their goal for the time period and how far along that goal they are.

Below that is a section for any upcoming challenges. By placing this on the homepage the user can see them and be enticed to join them, without needing to perform extra clicks to navigate to the search pages.

At the very bottom is the navigation bar. This will stay consistent throughout the whole application. It will highlight the corresponding icon of the page the user is currently on. While on the homepage there is also a create button that can be used to create challenges, teams, streaks and log an activity.

Search



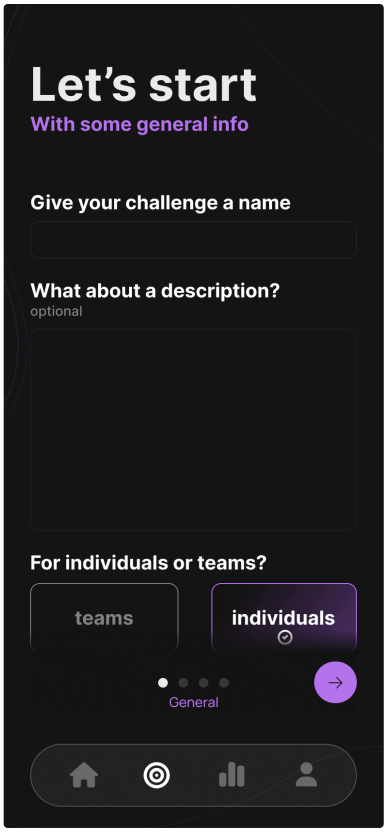


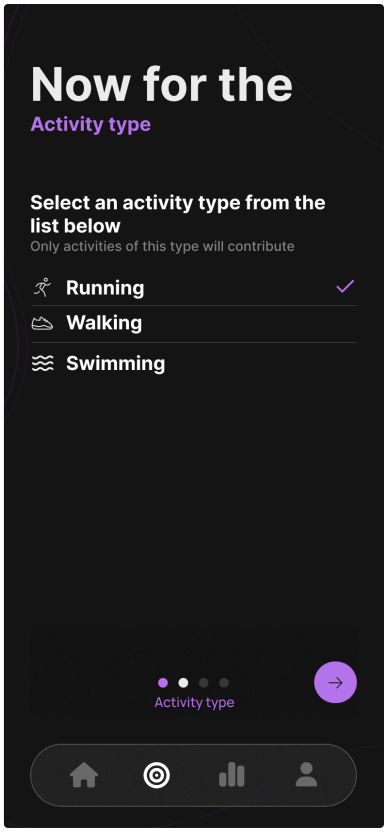
The create pages are mostly the same. They both have their search bar near the top of the page accompanied by their filters and sorting options. When selecting a filter it will show up in a pill next the the *filter* button that can be cleared by pressing the x icon. Whenever the user filters, sorts or searches new search results will pop up. They are displayed in a card format.

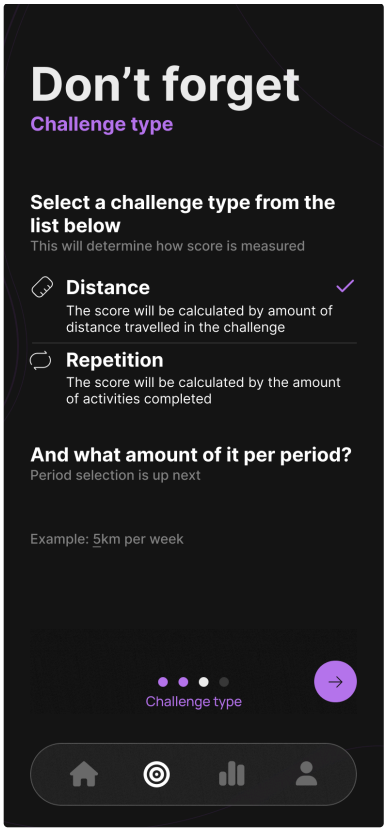
The cards contain some general information like title/name, description and activity type. Furthermore there's some info related to either a challenge or team. There's a join or leave button in the bottom right corner.

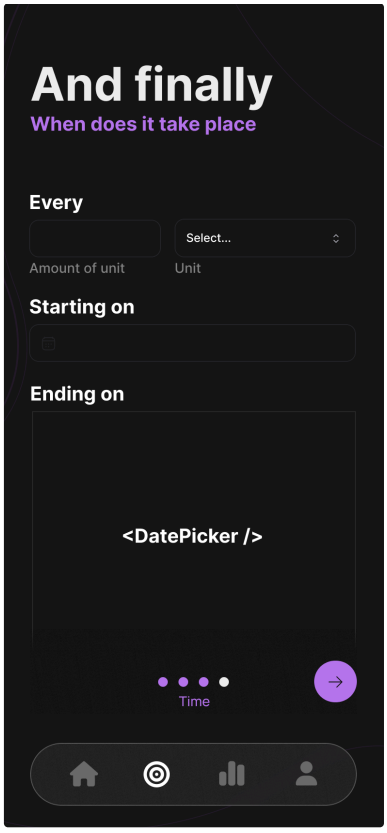
Create flow (challenge)

For the creation of challenges, streaks, teams and logging an activity we wanted to have multi-step forms that flow like a sentence. We didn't want the classic 2010 inspired forms. The create pages also have a purple theme instead of the bright yellow/green throughout the rest of the application.

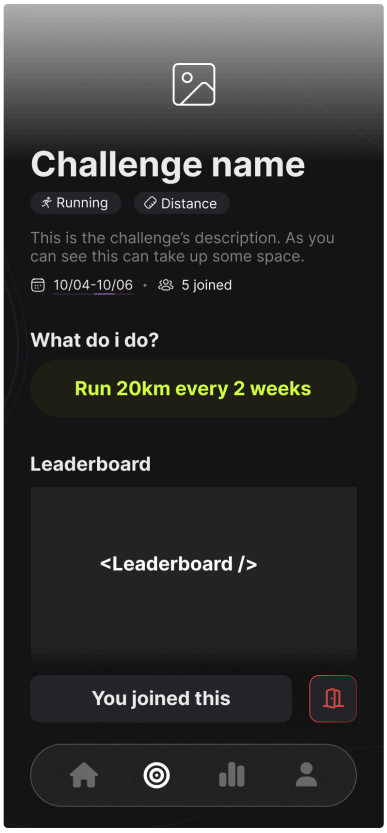

Let's start
With some general info
Give your challenge a name
What about a description?
optional
For individuals or teams?
teams
individuals
General


Now for the
Activity type
Select an activity type from the list below
Only activities of this type will contribute
Running
Walking
Swimming
Activity type


Don't forget
Challenge type
Select a challenge type from the list below
This will determine how score is measured
Distance
The score will be calculated by amount of distance travelled in the challenge
Repetition
The score will be calculated by the amount of activities completed
And what amount of it per period?
Period selection is up next
Example: 5km per week
Challenge type

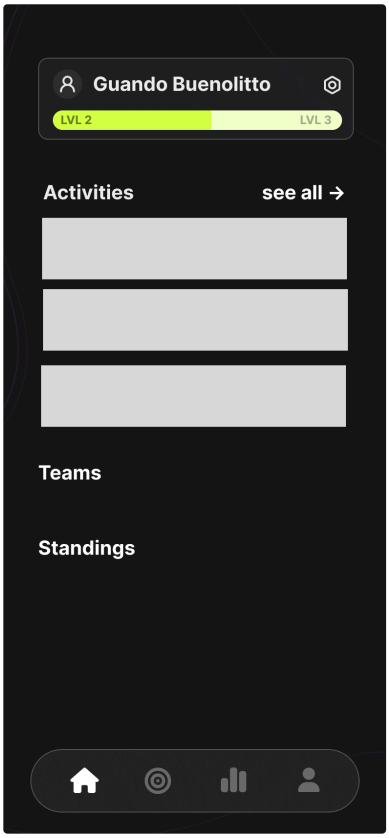

And finally
When does it take place
Every
Select...
Amount of unit
Unit
Starting on
Ending on
<DatePicker />
Time

Detail page



The detail page contains a cover image at the top. For the rest of the page it displays some information about the challenge or team.

Profile



The profile page contains a glass morphism card at the top displaying the users first and last name, accompanied by their profile image and level. In the top right is a settings icon that, when clicked expands the card to show some settings.

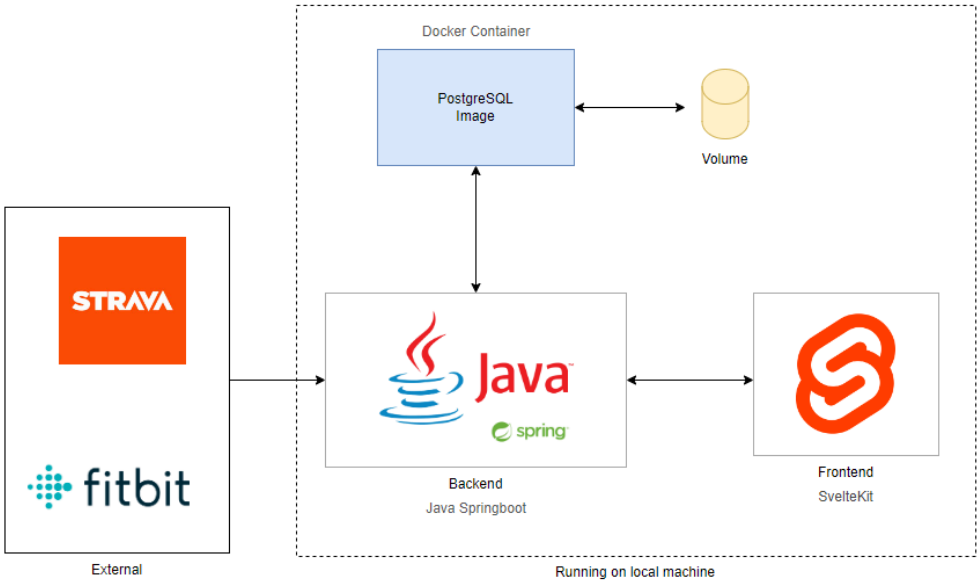
The profile page contains 5 of the user's most recent activities, their teams and some standings in leaderboards.

Project Architecture

This page contains some high level explanations on our project architecture. For more in depth information on each part, refer to their individual pages.

Technologies

Below is a schema showing how we set up our local environment for development. We did also deploy our frontend and backend so it'd be accessible online too.



Schema for local development

Backend

Our backend is a Java Spring API. It will handle incoming requests from the frontend and communicate with the database and external APIs.

We chose for a Java API since we've already done some projects with this in school, so we're familiar with it. A Java Spring combination is used in many project and has plenty of resources, documentation and tutorials to help us if needed. It is a very mature framework. Furthermore, Elision also uses this to write their backends. This means we can get plenty of feedback and inspiration from their developers. It was also required to use Java from Elision out for some of these very reasons.

Database

As database option we've opted for PostgreSQL. It's an open source project meaning we wouldn't have to pay any licenses for hosting one online. It's also a popular and much supported database option that we've used before in school or individual projects. In the end the choice for database doesn't really matter too much since we will not be dealing with massive amounts of data, very high performance needs or any of these things. When hosting, compute would be our bottleneck not the database itself.

Locally we're just running PostgreSQL in a Docker container.

Frontend

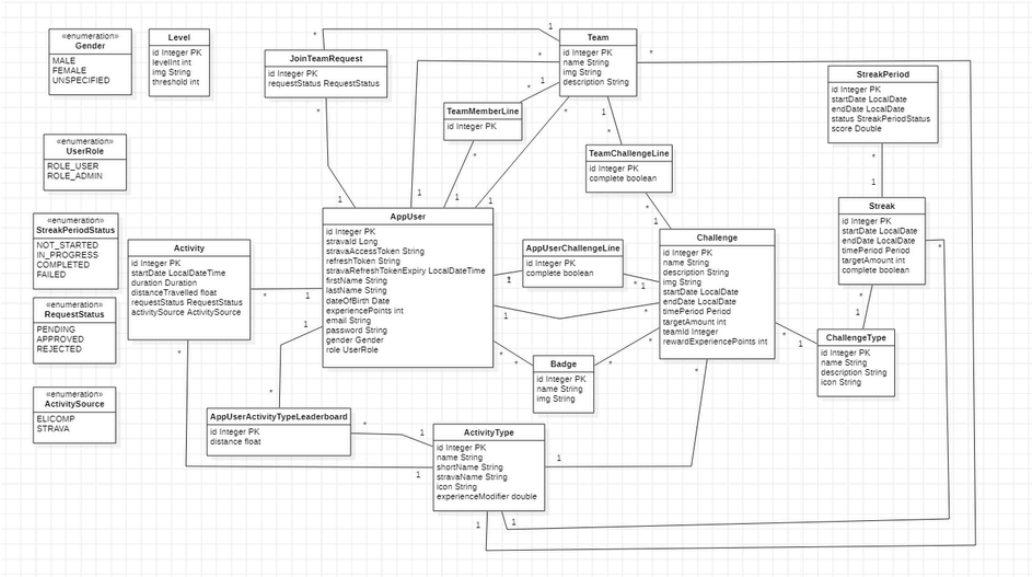
For frontend we were free to choose any option. We just had to keep in mind that at Elision they use NextJS so we could get some feedback on our code if we chose it. It was our first option, but in the end we settled for SvelteKit. Its a framework that’s been out for a while but is recently gaining more popularity. At Elision they are also looking into maybe using it in some of their future projects so building our project in this is a great test.

External

In the application users have the possibility to link their Strava account to our Elicomp application. This way any activities they log in Strava, will also be logged automatically in our database. In order to achieve this we need to use Strava as an external provider for authentication and resource access.

Data Model

Below is the ERD (Entity Relationship Diagram) for our database. Underneath the ERD you can find explanations for each table inside of the data model.



ERD

Tables explained

Table	Description
AppUser	Refers to the users of the application. Stores information regarding their login details, personal details and activity information.
Team	An entity which is comprised of a group of users. Teams are created with a specified activity type so users can participate in team based challenges.
Activity	Activities which the user logs are stored in the database to calculate the users progress in both challenges and the leveling system.
ActivityType	ActivityType refers to the type of the physical activity which the user has performed or the type of the activity for which a challenge is created ex. running, swimming etc..
Challenge	A challenge is defined with a start date, an end date and a target goal. The participants will be rewarded bonus rewards upon succesful completions of challenges.
ChallengeType	Challenge types are used to keep track of challenge progress for either distance type challenges/streaks

	and repetition based challenges/streaks.
Streak	A streak is a personal challenge a user can set for themselves, these streaks do not provide rewards.
StreakPeriod	Streak periods are what comprise a streak object. Streak periods are used to keep track of progress and completion for streaks.
JoinTeamRequest	Join team requests are created when a user requests to join a team, on approval they can be admitted to the team.
Badge	(Exists in database but not implemented in the reward system.) Badges are part of the rewards system in which users can earn them for accomplishing certan milestones.
Level	Levels are created on DB initilization. They are the “goal of the game“. Any experience points are used to level up and keep track of a users completion of the application.

OAuth & Strava

In our project we use the Strava API and Strava OAuth to authenticate a connection with Strava and to be able to act on our user’s behalf. Users can choose to connect their Strava account so their activities are automatically synced between Strava and our application. This section will explain how and why we used OAuth and go over the Strava integration.

OAuth

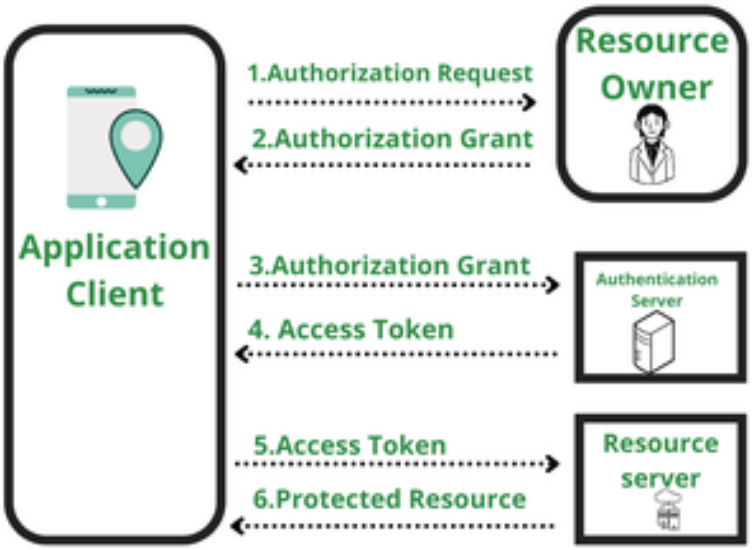
OAuth or Open Authorization is an open-standard authorization protocol or framework that provides applications the ability for “secure designated access”. This is better explained using an example: In our app users can choose to connect with their personal Strava account in order to more easily log activities. When the user clicks a button on our page that leads to the Strava login page, the OAuth process has started. During the login, they’ll be able to select scopes. Scopes are what the applications that make use of the OAuth provider need in order to access a scope of the user’s data. In Elicomp we request the scope `activity:read_all` since we need access to read the user’s private activities. Once the user finishes the OAuth process on Strava’s login page, they get redirected to an endpoint on our site that completes the flow.

OAuth is basically a way for companies (Google, Strava, Spotify,…) to provide applications (like ours) with access tokens so it can act on a user’s behalf. This is also often used for providing a simple and quick way of registering for an application: “Log in with Google”.

Flow

This image shows a very high level flow of OAuth 2.0. On the left hand side is our application, Elicomp, and on the right side would be Strava.

1. The frontend redirects the user to Strava’s login page. They log in successfully and provide access to the requested scopes.
2. We use the authorization token we get from Strava and make a request to their OAuth service with some identifiers (so they know it’s Elicomp making the request) and the received auth token. We then get back an access and refresh token which we can use to access resources.
3. In order to access a resource on the user’s behalf we need to include the access token in the header of the request.



High-level schema

Strava’s OAuth flow

Considerations

Token expiry

Due to security reasons both the access and refresh tokens expire after a while. For Strava's access token this is 6 hours. So once those 6 hours are over we should make use of the refresh token to request a new access token. If a new access token is requested, a new refresh token *could* also be provided by Strava. If the received and stored refresh tokens do not match, the newly received one should be used for any future requests.

⚠️ Currently the application does not do this. If the access token expires the user will simply have to go through the process again.

User doesn't provide correct scopes

During the flow the user can choose the scopes they allow the application access to. If they do not accept to provide access to our required scopes the application should show them an error message indicating they need to provide the correct scopes.

⚠️ Currently the application does not do this.

Which providers do we use?

Strava 🔗

Endpoints

List Athlete Activities

Scopes

`activity:read_all`

Other possible providers

Fitbit 🔗

Resources

https://developers.strava.com/docs/authentication/

Strava integration

As mentioned in the OAuth document, we'll make use of a Strava connection to act on our user's behalf and request their latest activities. In this document the flow of how this all plays out is described into detail.

Why integrate Strava?

As Strava is already an established application for logging exercises, we wanted to cater to the regular habits of our users. By making it so they can seamlessly transfer their activity data to the EliComp application, we hope to remove any unnecessary hassle in using the application. For users who don't have or don't use Strava, they will still be able to manually log their activities in a similar manner. Overall the integration of the Strava API will remove any redundant or duplicate activity logging from users, thus not pushing them away from the application.

Webhook

Strava gives us the option to subscribe to events for our application. We will make use of this since the Strava API limits us to a certain number of requests per day, we won't have to make needless requests and waste our quota. Below you can find step by step instructions on how to set this up.

1.) We first need to create an endpoint in our backend so that Strava can make POST requests to it, and also so it can make a singular GET request during the start of the subscription.

1.1) First let's set up the endpoint which Strava will use to validate the callback url we will share with it when we make a request to create a subscription. This endpoint will reply to strava with the hub.challenge field to verify that it is indeed a valid endpoint.

```
@GetMapping
    public ResponseEntity<Map<String, String>> verifyWebhook(@RequestParam(name = "hub.challenge") String hubChal
        Map<String, String> response = new HashMap<>();
        response.put("hub.challenge", hubChallenge);
        return ResponseEntity.ok().body(response);
    }
```

2.) Now we need to also make another endpoint so that Strava can make POST requests with the logged activities of our authorized users.

```
@PostMapping
    public void receiveStravaNotification(@RequestBody StravaEventPush request) throws ActivityTypeNotFoundExcept
        stravaService.createNewActivityOnEventPush(request);
    }
```

2.1) Strava's event notifications have the following format:

object_type string	Always either "activity" or "athlete."
object_id long integer	For activity events, the activity's ID. For athlete events, the athlete's ID.
aspect_type string	Always "create," "update," or "delete."

updates hash	For activity update events, keys can contain "title," "type," and "private," which is always "true" (activity visibility set to Only You) or "false" (activity visibility set to Followers Only or Everyone). For app deauthorization events, there is always an "authorized" : "false" key-value pair.
owner_id long integer	The athlete's ID.
subscription_id integer	The push subscription ID that is receiving this event.
event_time long integer	The time that the event occurred.

The only fields here that are relevant in the scope of our application are:

- object_type
- object_id
- aspect_type
- owner_id

2.2) Since we will be using strava only to receive new activities for our athlete, we will filter the incoming requests and only process the ones where:

- object_type = “activity“
- aspect_type = “create“

3.) Now we can set up the logic of what to do whenever a new event is posted to our endpoint.

```
public void createNewActivityOnEventPush(StravaEventPush stravaEventPush) throws IOException, InterruptedExcepti
        if(stravaEventPush.getObject_type().equals("activity") && stravaEventPush.getAspect_type().equals("creat
            Long stravaId = stravaEventPush.getOwner_id();
            Long activityId = stravaEventPush.getObject_id();

            AppUser user = userRepository.findByStravaId(stravaId);
            String userAccessToken = user.getStravaAccessToken();

            HttpRequest request = HttpRequest.newBuilder()
                   .uri(URI.create(stravaActivityUrl + activityId))
                   .header("Authorization", "Bearer " + userAccessToken)
                   .method("GET", HttpRequest.BodyPublishers.noBody())
                   .build();
            HttpResponse<String> response = client.send(request, HttpResponse.BodyHandlers.ofString());

            ObjectMapper objectMapper = new ObjectMapper();
            StravaActivityResponse activityResponse = objectMapper.readValue(response.body(), StravaActivityResp

            ZonedDateTime zonedDateTime = ZonedDateTime.parse(activityResponse.getStart_date_local(), DateTimeFo
            LocalDateTime localDateTime = zonedDateTime.toLocalDateTime();

            ActivityType activityType = activityTypeRepository.findByStravaName(activityResponse.getSport_type()
            ActivityDto activityDto = ActivityDto.builder()
                    .activityTypeId(activityType.getId())
                    .startDate(localDateTime)
                    .duration(Duration.ofSeconds(activityResponse.getMoving_time()))
                    .distanceTravelled(activityResponse.getDistance()/1000)
```

```
                    .build();
            activityService.createActivityOnStravaEventPush(activityDto, user, activityResponse.isManual());
        }

    }
```

4.) Finally, we can now send a request through curl or postman to tell Strava we want to subscribe to these notifications, we can do this with the following request template:

```
$  curl -X POST https://www.strava.com/api/v3/push_subscriptions \
      -F client_id=5 \
      -F client_secret=7b2946535949ae70f015d696d8ac602830ece412 \
      -F callback_url=http://a-valid.com/url \
      -F verify_token=STRAVA
```

4.1) The subscription request if successful will return a subscription_id which will be important to save so that in the future if we want to cancel our subscription we can do so easily with the following request:

```
$ curl -X DELETE "https://www.strava.com/api/v3/push_subscriptions/12345?client_id=5&client_secret=7b2946535949ae
```

Conclusion

Strava's API being added to the application makes it so users will have an easier time keeping track of their activities and progress without having to manually log their activites in both applications. Once an activity is logged in Strava, it will automatically trigger the flow to be processed in the EliComp application as well.

Backend

The backend of this application is a Java Spring Boot API. Requests coming from the frontend will be handled and the necessary database operations will be made. Java was selected for this project as we have prior experience in using it from our school courses and projects. Java Spring being an established framework with an abundance of documentation available made it the obvious choice as well. As Elision also uses Java Spring to program their backends, our mentor was also able to guide us more easily in case we ever had any issues.

Java

Java is an open-source programming language which is designed around Object-Oriented Programming (OOP) principles. OOP principles promote modular, reusable and maintainable code, making Java a great choice for development.

As we will be using personal information from users in this project, it is important to keep this information secure. Java has many built-in security features which we can make use of depending on our needs.

In conclusion, as we want our application to:

- perform well
- be secure
- be scalable
- be maintainable

We will be programming our backend in Java.

Spring

Spring is an open-source framework built with Java. With Spring, we can create modular, reusable, and maintainable code, which is crucial for long-term development and scalability.

As mentioned in the Java section, security is something we have to take into account. Spring offers diferent security features that can be tailored to meet specific requirements, ensuring that personal data remains protected against unauthorized access.

In summary, considering the above mentioned requirements of our application, Java along with Spring is our choice for backend development.

API Endpoints

User

REST Method	Endpoint	Description
GET	API/Users/{id}	Retrieves the requested User
DELETE	API/Users/{id}	Deletes the requested User
PUT	API/Users/{id}	Updates information for the requested User
POST	API/Users	Creates new User

Team

REST Method	Endpoint	Description
GET	API/Teams/{id}	Retrieves the requested Team
DELETE	API/Teams/{id}	Deletes the requested Team
PUT	API/Teams/{id}	Updates information for the requested Team
POST	API/Teams	Creates new Team

Challenge

REST Method	Endpoint	Description
GET	API/Challenge/{id}	Retrieves the requested Challenge
DELETE	API/Challenge/{id}	Deletes the requested Challenge
PUT	API/Challenge/{id}	Updates information for the requested Challenge
POST	API/Challenge	Creates new Challenge

Badge

REST Method	Endpoint	Description
GET	API/Badge/{id}	Retrieves the requested Badge
DELETE	API/Badge/{id}	Deletes the requested Badge
PUT	API/Badge/{id}	Updates information for the requested Badge
POST	API/Badge	Creates new Badge

Activity

REST Method	Endpoint	Description
GET	API/Activity/{id}	Retrieves the requested Activity
DELETE	API/Activity/{id}	Deletes the requested Activity

ActivityType

REST Method	Endpoint	Description
GET	API/ActivityType/{id}	Retrieves the requested ActivityType
DELETE	API/ActivityType/{id}	Deletes the requested ActivityType

PUT	API/ActivityType/{id}	Updates information for the requested ActivityType
POST	API/ActivityType	Creates new ActivityType

Level

REST Method	Endpoint	Description
GET	API/Level/{id}	Retrieves the requested Level
DELETE	API/Level/{id}	Deletes the requested Level
PUT	API/Level/{id}	Updates information for the requested Level
POST	API/Level	Creates new Level

Tools & Libraries

This project uses Java and the Spring Framework to build a strong and efficient backend application. By using different tools and libraries, we ensure our app is easy to maintain, scalable, and performs well. This page will go over these used tools and libraries.

Java

Java is a powerful, object-oriented programming language that can run on any device with the Java Virtual Machine (JVM). This makes it a great choice for our project because we can deploy it on different platforms without any issues. Java also manages memory automatically, which helps us avoid memory leaks. It has a rich library with lots of useful features, making development faster and easier.

Spring Framework

The Spring Framework provides a comprehensive model for building Java applications. It has a core feature called the Inversion of Control (IoC) container, which manages the lifecycle of Java objects, known as beans. This helps in making our code more modular and easier to manage. Spring also supports Dependency Injection (DI), which means that it injects dependencies into our classes at runtime, making our application easier to test and maintain. Additionally, Spring can integrate with other frameworks like JPA and RESTful web services, giving us a lot of tools to work with.

Java Persistence API (JPA)

JPA is a standard way to manage relational data in Java applications. It helps us map Java objects to database tables, making database interactions much simpler. JPA uses entities to represent data stored in a database. Overall JPA makes it easier to perform operations like creating, reading, updating, and deleting data. It also provides a query language called JPQL, which lets us write database queries using Java objects.

Lombok

Lombok is a library that helps us write less boilerplate code. By adding annotations to our classes, Lombok generates common methods like getters, setters, and constructors automatically during compilation. This makes our code cleaner and saves us from writing repetitive code.

PostgreSQL Driver

The PostgreSQL JDBC Driver allows our application to connect to our PostgreSQL database. It's a pure Java implementation, ensuring it can run anywhere. It fully supports JDBC standards, allowing us to execute SQL queries, retrieve results, and manage database transactions. The driver also supports SSL encryption for secure data transmission. It works with multiple versions of PostgreSQL, ensuring compatibility.

JUnit and Mockito

For unit testing, we used JUnit and Mockito. These tools helped to ensure our code is working correctly and made it easier to catch bugs early.

JUnit is a testing framework for Java that provides annotations and assertions to write and run tests. It allows us to create test cases for our methods and verify their behavior. By running these tests frequently, we can ensure that changes to our code do not break existing functionality.

Mockito is a mocking framework that works well with JUnit. It allows us to create mock objects and define their behavior. This is particularly useful when we want to test a class in isolation and avoid dependencies on other parts of the application. By mocking dependencies, we can focus on testing the logic of the class itself.

Using JUnit and Mockito together, we can write comprehensive unit tests that verify the behavior of our code under different conditions. This improves the reliability of our application and helps us maintain high-quality code.

Additional Dependencies

Our project also uses other dependencies such as:

- **Spring Boot Starter Data JPA**: Simplifies database access and interactions using JPA.
- **Spring Boot Starter OAuth2 Client**: Makes it easy to add OAuth2 authentication and authorization.
- **Spring Boot Starter Web**: Supports web application development, including RESTful services.
- **JSON Web Token (JWT) Libraries**: Includes jjwt-api for core JWT functionalities, jjwt-impl for runtime support, and jjwt-jackson for JSON serialization and deserialization using Jackson.
- **Spring Boot Starter Test**: Provides comprehensive testing support for Spring Boot applications.
- **Spring Security Test**: Helps in testing Spring Security features.
- **Hibernate Validator**: Ensures bean validation.
- **Apache Commons Lang3**: Offers additional utility methods for Java classes.

Authentication

In the project we implemented basic email and password based authentication. The frontend sends over the combination and the backend checks for its validity against the records in our database. If it is valid, a JWT gets returned to the frontend. This token can be used to authenticate the current user. This way they don't have to login again on every request.

Key Terms

Authentication involves checking user identity with provided credentials. It answers the question, "Who are you?" **Authorization** determines what actions a user can perform, answering the question, "What can you do?" The **principal** refers to the authenticated user, while **granted authority** describes the permissions of that user. A **role** is a collection of permissions assigned to the authenticated user.

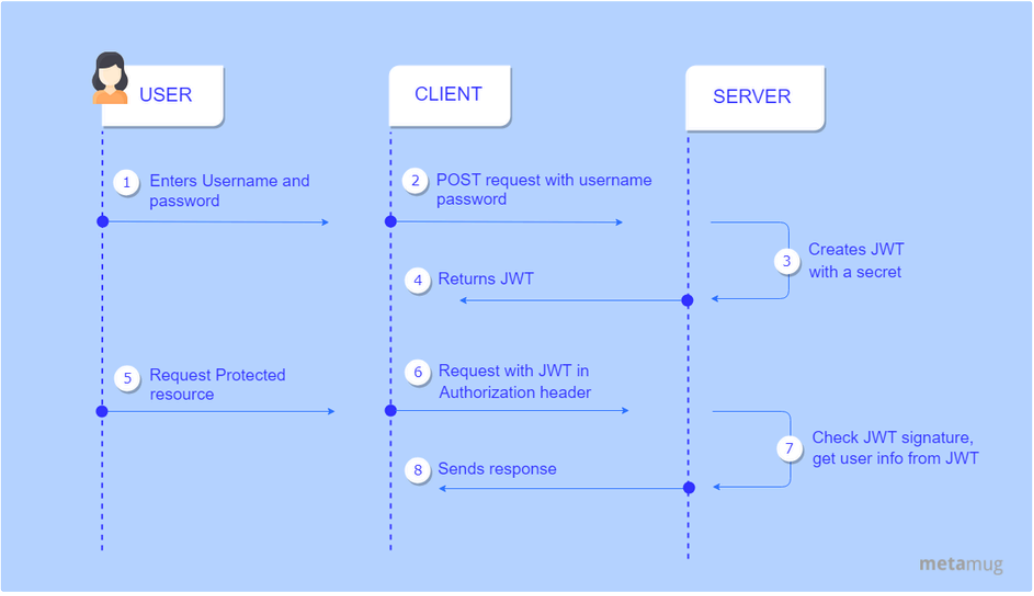
Spring Security provides authentication and authorization for Java applications. It supports various authentication methods such as HTTP Basic, Form-based, and JWT. JWT (JSON Web Token) is a compact way of representing claims transferred between two parties. It consists of a header, payload (which contains user information), and a signature. A JWT Filter intercepts incoming requests to extract and validate the JWT before confirming the user's identity.

How It Works

During authentication, when a user logs in, Spring Security checks their credentials. If the credentials are correct, a JWT is generated with user details and signed with a secret key. For subsequent requests, the JWT is included in the authorization header. The server verifies the token’s signature and user information to grant access. Since JWTs are stateless, the server doesn't need to store session information.

Benefits of JWT

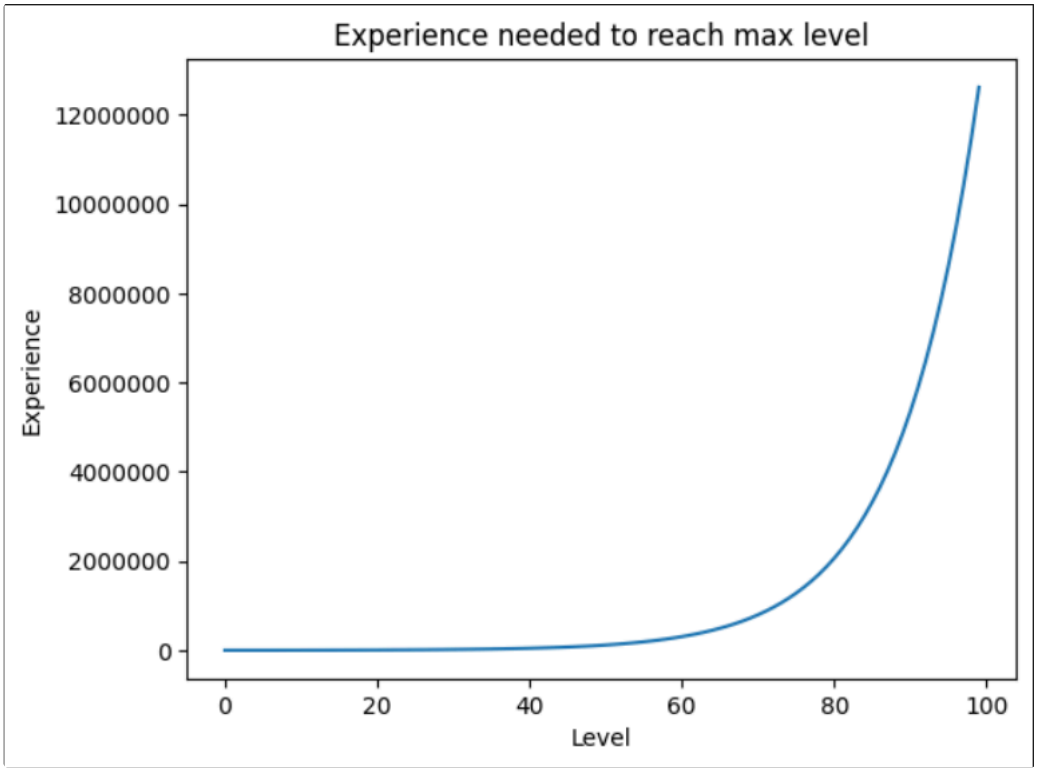
Enhanced Security is achieved through digitally signed JWT tokens, which prevent unauthorized access. Scalability is simplified as the stateless nature of JWTs allows for easy session management and horizontal scaling, with each server able to verify tokens independently. Simplified Integration with Spring Security requires minimal setup, and JWTs can be validated using standard libraries. Performance is improved because JWTs are self-contained, storing all necessary user information, eliminating the need for database queries or maintaining server-side session states during authentication.



Experience System

The experience system will be used to incentivize users into using EliComp in the longer run. We wanted to treat EliComp as a “game“ which users could finish in a sense. By setting a maximum level of 100 and an exponential experience requirement to level up, we hope to induce the feeling of a fast progression at the beginning with a slower more “grindier“ experience towards the latter end of progression. With this we hope both average users and more physically intense user will have an enjoyable experience.

Determining the experience thresholds



The experience threshold formula we decided to use is as follows:

```
levels = {}

for i in range(100):
    if i == 0:
        levels[i] = {"level": 1,"threshold": 0, "diff": 0}
    elif i == 1:
        levels[i] = {"level": 2, "threshold": 100, "diff": 100}
    else:
        threshold = levels[i-1]["threshold"] + levels[i-1]["diff"]+ round(levels[i-1]["diff"]*.1)
        levels[i] = {"level": i+1, "threshold": threshold, "diff": threshold - levels[i-1]["threshold"]}
```

The above formula was inspired by the popular MMORPG runescape. This curve allows an “illusion“ of quick progress in the beginning, hooking in new and less motivated users with the dopamine hit of fast levels. But the thresholds increase exponentially, creating an interesting breakpoint where a user can hit level 50, the “halfway“ point to max level in 1-2 months but in reality the literally halfway point to level 100 is level 94 in terms of experience threshold. The beauty behind this is that every user will feel rewarded, whereas the hardcore fitness enthusiast will find the true challenge in hitting maximum level.

We wanted to have a reasonable time to completion of the game for an average employee at Elision. From the employees we questioned, the average tenure at Elision was around 7 years long. After searching online we also concluded that the average amount of physical activity a person performs is walking 5 times a week at a pace of 5 km/h.

Experience modifiers for different activity types

Since users can also perform different activates, we used the following formula to determine experience modifiers for each activity type:

```
activityTypes = {
    "walking" : {"speed": 5, "modifier": 0},
    "cycling" : {"speed": 19, "modifier": 0},
    "running" : {"speed": 8.5, "modifier": 0},
    "swimming" : {"speed": 3.5, "modifier": 0}
}
def calculateBaseModifier(speed):
    return 5 / speed
for activity in activityTypes:
    activityTypes[activity]["modifier"] = calculateBaseModifier(activityTypes[activity]["speed"])
```

we used the speed of walking as the base for our calculations as it is the most commonly performed “exercise“ in this context.

Experience gain formula

The experience gain rate was calculated to ensure a linear progression, we didn’t want activities or challenges to be less rewarding as time went on, we wanted users to have to do more in order to progress. Remember, the main goal of EliComp is to promote physical fitness and movement, not to blaze your way to level 100.

The formula was tinkered with over and over to get an appropriate amount of session needed to reach max level, in our case this ended up being 1253 session of the predefined average exercise session (a 30 minute walk at a pace of 5 km/h):

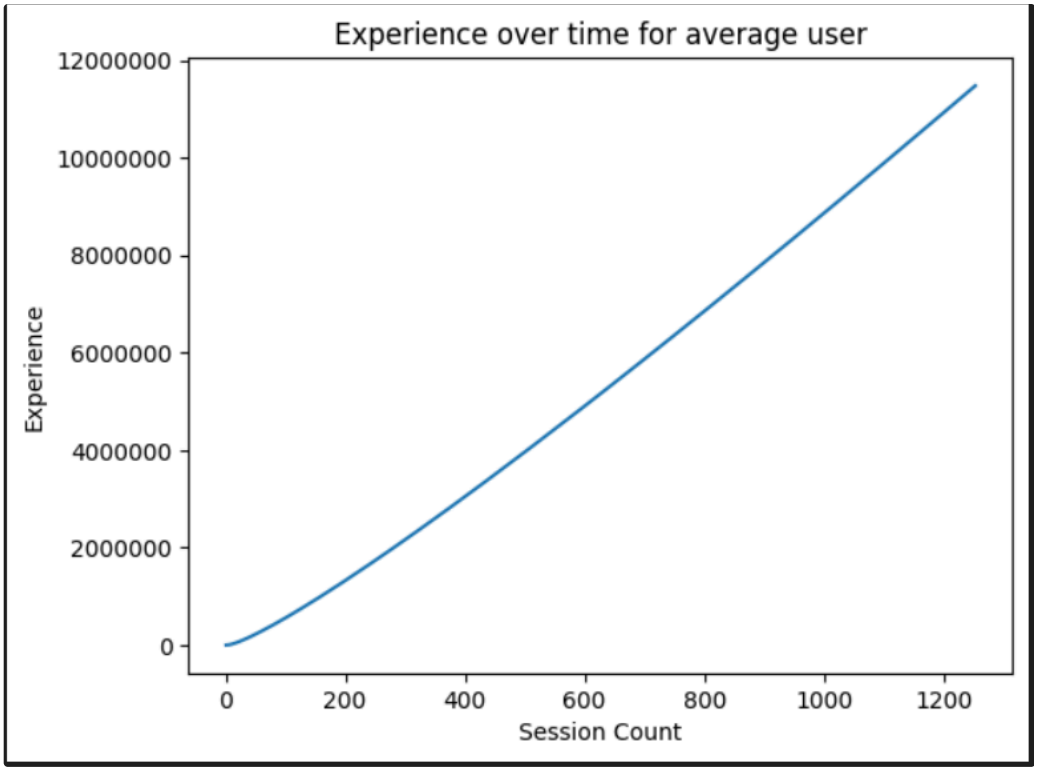
```
def calculateExperienceGained(durationInMinutes, distanceInMeters, activityType, level, teamSize):
    experience = ((distanceInMeters / durationInMinutes) * activityTypes[activityType]["modifier"] * level + leve

    if teamSize > 2:
        experience *= 1.2

    return experience
```

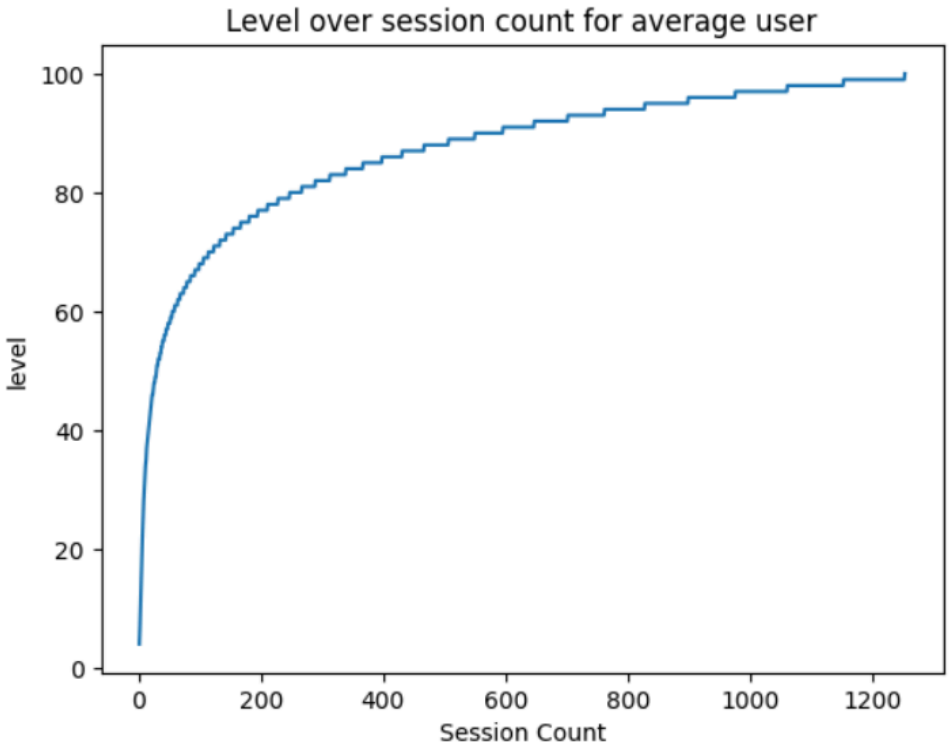
```
# function to have user walk 5x a week at a pace of 5 km/h for 30 minutes each time until max level
averageUserExperience = []
averageUserSessionCount = []
averageUserLevel = []
def walkTillMaxlevel(user):
    counter = 0
    while getCurrentLevel(user["experiencePoints"]) < 100:
        counter += 1
        averageUserSessionCount.append(counter)

        user["experiencePoints"] += calculateExperienceGained(30, 2500, "walking", getCurrentLevel(user["experie
        averageUserExperience.append(user["experiencePoints"])
        averageUserLevel.append(getCurrentLevel(user["experiencePoints"]))
    return counter
```



Summary

With this system and reasoning in place, we were able to create an experience system which allowed for an average user to reach max level in a time frame of 3-4 years. This is an acceptable length in our opinion as we want to promote physical fitness over a long duration. With the thresholds, experience gain rate and reasoning that we provided, users can expect an experience like the graph below:



We hope this system will provide a constant sense of improvement and accomplishment for progressing their level and “beating“ EliComp.

Frontend

Our frontend is written in SvelteKit. This page and its subpages go over why we chose this and what exactly Svelte is all about.

Svelte and SvelteKit

Although their names are much alike, they do not serve the same purpose. Svelte is a component-based UI framework (think of React). It's also reactive, meaning changes in state automatically update the UI. One of the biggest features of svelte however is its compiler. Svelte code is written in `.svelte` files and they are compiled at build time to JS.

SvelteKit on the other hand is built on top of Svelte and is Svelte's Full Stack framework. It supports multiple features like SSR (Server Side Rendering), file-based routing, form actions, API routes and much much more.

Svelte

Svelte is a component-based JavaScript UI framework created by Rich Harris, released in 2016. It has garnered a lot of attention since then and is loved by many developers for its syntax and speed.

"What is Svelte" from Vercel

Compiler

Svelte uses a compiler to compile the `.svelte` code to produce optimized JS code, allowing for lightweight and fast single or multipage web apps. In comparison to React, Svelte does not make use of a virtual DOM to keep track of *if a state changes, what should be updated?*

React's virtual DOM is basically a virtual representation of the UI kept in memory and synced with the real DOM. In order to show to the user the updated state React first needs to know what has updated. It does this by going through 3 steps. Instead of performing those 3 steps sequentially, Svelte compiler knows at *build time* how things could change in your app, rather than waiting to do the work at *run time.*

> React is also switching to a compiler. This will make writing performant applications easier. It's not recommended for production yet, but is in production use at Instagram and other parts of Meta.

Component based

Svelte is a component based framework. This means the logic gets split into components, building blocks. Using a component based workflow brings with it that a developer writes less code due to reusability, resulting in clearer code to read.

Reactivity

Svelte has a powerful system of *reactivity* for keeping the DOM in sync with your application state. It also has some neat shortcuts for the DX. Vercel has some great documentation on Svelte's reactivity showcasing how assignments, declarations and statements work.

For example, in React state and reactivity is done using the `useState` hook: `[count, setCount] = useState(0)`. In Svelte you can just use `let count = 0` . Under the hood Svelte does a lot of processing to make this work.

The usage of `let` brought along some trickiness in trying to figure out which values are reactive in larger and more complex applications. This all changes with Svelte 5.

Svelte 5 and runes

In Svelte 5 `let` is no longer the way to handle reactivity. This should now be done using runes. For the developer experience not much has changed except for a few syntax changes and harder to find documentation. Under the hood however, many changes were made resulting in even more performant apps.

In our project, we made use of Svelte 5. A more in-depth comparison between version 4 and 5 can be found in the upcoming section “Svelte 4 vs 5“.

> Svelte 5 is currently in beta and backwards compatible with Svelte 4. Applications using `let` shouldn't break. More information about Svelte 4 vs 5 can be checked out on the “Svelte 4 vs 5“ section.

Built in modules

Svelte provides some built in modules that provide some quality of life improvements. Some of these are listed below. The others can be checked out on the Svelte docs.

Store

The store module allows for keeping track of state across a whole application. On a high level, you can compare this to Jotai or Zustand for React. You’d define a variable in a store (a file, ex: `store.ts`) and then any page or component in the application can access and/or modify this variable. It’s not bound to a single component.

A shopping cart is a good example for this. It’s something that should be kept track of throughout the whole site. You don't want your cart to suddenly be empty if you leave a product’s page. This could also be achieved by passing the state as a prop to components, but this results in prop drilling.

> This state is of course lost if the user leaves or refreshes the site.

Transition

Page transitions can be a good way to make your application feel more complete and provide a smoother UX. Svelte has a transition module, which makes this extremely easy.

SvelteKit

As said earlier, SvelteKit is Svelte’s fullstack framework. In comes with file based routing, SSR, prerendering, best practices for web apps, form actions and more.

File based routing

SvelteKit supports file based routing out of the box. When using file based routing every single route is defined in a specific folder in the project hierarchy. For example for a `/foo` page to exist on the site, you'll need to add (in SvelteKit’s case) a `foo` folder to the `src/routes` folder. This `foo` folder should contain at least a `+page.svelte` file. This Svelte file will then contain the html and/or logic. There are some more available files all listed here.

You’re not just limited to just pages either. There’s also the possibility to create your own API routes, but since we’ll be doing our backend in Java this wont be necessary. The only endpoint we need as an API route in our frontend is the callback used during the Strava OAuth process.

Form actions

Form actions are functions that are called by the client and ran on the server. For example when you change some details about your account and press the submit button, a form action is called and ran on the server. This action could then check if the data is adhering to a validation schema and if it is, make an API call to our backend.

Why form actions

SvelteKit encourages the use of form actions because they work without having JS enabled, providing a better experience for some users. Another advantage is also keeping your backend hidden from the end user, because it's the server making requests not the end user's browser.

SSR and prerendering

Prerendering

When using prerendering on a page, an HTML file gets generated for it at build time. This could be used for a landing page where information about the site is displayed. Any page not relying on dynamic data (for example a user profile, which is different for every user) is a perfect candidate for prerendering. It can also be used for pages *with* personal dynamic data, but then the data will need to be client fetched. When setting prerender on a page it tells SvelteKit to not include the code to render these pages on the server.

SvelteKit Glossary - prerendering

Server side rendering (SSR)

By default, SvelteKit renders the page on the server first so the HTML gets generated on the server. It also runs the load function from the +page.server.ts and any code in the <script> tag. The HTML, CSS and Javascript then gets sent to the client. If there's any data being fetched during SSR, by default SvelteKit will store this data and transmit it to the client along with the server-rendered HTML. So then the client has the HTML and JS but no interaction can happen just yet because the JavaScript isn't connected to the HTML yet. So next, Svelte will check that the DOM is in the expected state and attach event listeners in a process called hydration.

SvelteKit Glossary - Hydration

Hydration: What is it?

But why SvelteKit?

There are so many great frameworks out there: NextJS, SolidJS, Astro, just to name a few. So why did we pick SvelteKit?

Developer Experience

Svelte has a great DX. Dealing with state is nearly effortless. It has plenty of built in modules. Its HTML superset is easy to understand and fun to write in. It has Laravel like logic blocks in the HTML that make it clear what the code is doing, as opposed React's conditional rendering.

Of course each of the earlier mentioned (and other) frameworks have their own strong points. NextJS has some great caching capabilities, SolidJS's compiler makes for some of the best performance, etc…

Elision

Some of Elision's clients are looking into maybe using Svelte(Kit) for some of their projects. An application like ours is a great test run to find some of SvelteKit's strong and weakpoints.

Svelte 4 vs 5

On September 20, 2023 Svelte 5 got announced and it was sold as a major new version. Not minor like Svelte 4 was. Svelte 5 is a ground-up rewrite of the framework. There are plenty of syntax changes and although these can be annoying for existing users, new users will have a better understanding and better control of what their code is actually doing. This document will explore some of the changes made. All of these are explored in more detail in the official preview docs, where the other changes can be found too.

⚠ Svelte 5 is currently still in preview and not ready for production yet.

Runes

In Svelte 4 reactivity was done through the magic `let` and `$:` declarations. It's not clear what these do and so as a learner you just accept them for what they are. In Svelte 5 these are being replaced by the `$state`, `$derived` and `$effect` runes. Many people complained, but there is good reason for this change. There were bugs, syntax issues and just general shortcomings with the old approach. The new way makes things faster (according to official docs) and has a smoother learning curve for newcomers.

ℹ Objects and arrays are made deeply reactive.

Event handlers

In Svelte 4 event handlers the `on:` directive was used to attach an event listener to an element, like so: `<button on:click={}>`. In Svelte 5 it's a property like any other: `<button onclick={}>`. There are multiple reasons for this change, all listed out here. The most important reason is no more boilerplate code. This change does also bring along with it that event modifiers can no longer be added to handlers like they could in Svelte 4. An event modifier looked like this: `<form on:submit|preventDefault={handler}>` . These made it easy to quickly perform something. They don't give the actual reason for removal, but do make a great argument:

> Adding things like `event.preventDefault()` inside the handler itself is preferable, since all the logic lives in one place rather than being split between handler and modifiers.
>
> Svelte 5 preview docs - event handlers

Universal Reactivity

In Svelte 5, there is no more need for the `'svelte/store'` package as state can be placed in a separate `.svelte.js` or `.svelte.ts` module now, because it is part of the actual Svelte language. Take for example the following code:

```
let count = $state(0);

export function getCounter() {

    function increment() {
        count += 1;
    }

    return {
        get count() {
            return count;
        },
        increment
    };
```

```
15 }
```

Wherever this `getCounter` method gets imported, the `count` will be able to be incremented and read. Reactive state is not bound to the top level of a component anymore. It can easily be shared between components by extracting it into a file.

Snippets

With snippets reusable chunks of markup can be created inside of components. A pretty common use case for this can be found here.

With the introduction of snippets, `<slot />` should no longer be used. A `<slot>` tag was used to render out any children that got passed to the component. Think of a layout file that provides some layout that should be the same for all of its *children*. In Svelte 5 this has been replaced by a snippet: `{@render children()}` . The children also need to be retrieved from the props: `let { children } = $props();` . This does result in more code than a simple `<slot>` tag.

Issues

We don't know if these issues are Svelte 5 exclusive, but since we did have them happen, we'll note them down. Most of these aren't really big issues, but just annoyances that other frameworks do not have.

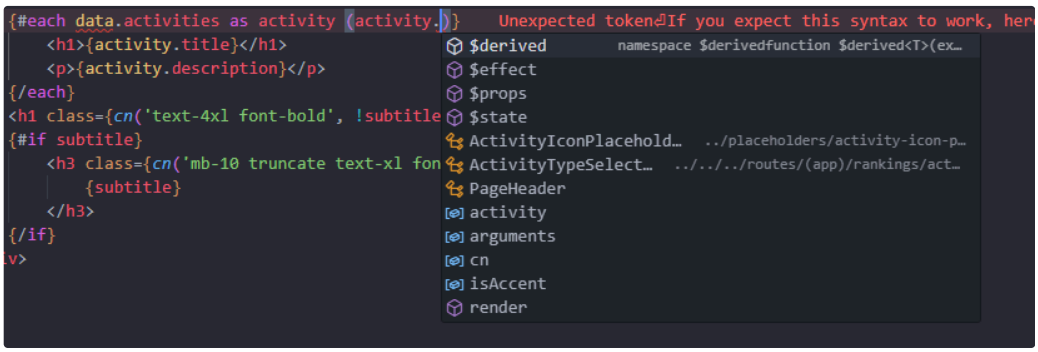
> At the time of writing this document and the code itself, Svelte 5 is still in beta.

Code completion

When templating the code completion doesn't really work all that great. Here's an example: I want to loop over all activities i got back from my API call. Then the following could be the code for that.

```
1 {#each data.activities as activity (activity.id)}
2   <h1>{activity.title}</h1>
3   <p>{activity.description}</p>
4 {/each}
```

An issue that would often occur when writing this code is the lack of context aware code completion. Most of the time when typing out the key (`activity.id`) there's no completion that is contextual to `activity` . It's all just general Svelte code like `$derived` or `$props` . The same thing happens when accessing attributes of variables inside of the HTML part of the `.svelte` file.



Hydration

Sometimes there are issues with hydration mismatch between server and client, even though there should not be any. When restarting the dev server these magically disappear.

Library compatibility

Even though Svelte 5 is backwards compatible with Svelte 4, some libraries do seem to break. Another thing we found is when installing some packages with `npm` it would cause the dependency tree to break, failing to start the application. When installing the same packages with `pnpm` everything worked just fine.

Documentation

This is more of a general Svelte issue, not just Svelte 5 related, but it's the lack of documentation. In our opinions the documentation provided on the Svelte 5 docs (and sometimes even Svelte 4) just isn't up to par with other some frameworks. This leaves us figuring out a lot of stuff ourselves. Sometimes the documentation is just plain missing explanation or the information comes across very unclear, forcing the reader to go through it multiple times.

Conclusion

Svelte 5 brings along some pretty substantial changes (primarily under the hood). It is, however, still in beta so some bugs and unfinished stuff come along for the ride. However, the positives outweigh the negatives and given enough time Svelte 5 will be loved by more and more developers.

Links

Svelte 5 preview docs

Svelte 5 announcement blog post

How does the Svelte compiler work (v4)

Findings of migrating to v5

NPM Packages

This page contains explanation on the packages that we use in our frontend and why we use them.

Shadcn-svelte

Shadcn-svelte is a collection of re-usable components. These can be copy and pasted or downloaded with the CLI straight into the hierarchy, not in the `node_modules` folder. What this means is that, as a developer, you have full control over every part of the component. It can be styled and/or animated to your likes.

So why not just write the complete component yourself? Shadcn-svelte is built on top of Melt UI and Bits UI. Melt UI provides a very low level builder API for creating headless components. It's Melt UI that contains the accessibility features for every component (keyboard navigation for example), which is a long and tedious process to get just right. Bits UI is built on top of Melt UI. Bits takes the Melt API and wraps it in a more familiar component interface. Shadcn-svelte then adds some default styling using Tailwind on top of that.

Tailwind

Compared to normal CSS, Tailwind is a blessing. It allows you to write CSS directly in the HTML. This can be done via the `class` attribute of an element, it will then look something like this: `class="p-2 text-center bg-blue-200"`. By writing the CSS directly in the HTML we don't have to come up with class names or deal with element selecting. When it comes to building the application for production, Tailwind scans all the HTML files, JavaScript components, or any other templates for class names, generating the corresponding styles and then writing them to a static CSS file. It aims to produce the smallest CSS file possible by only generating the CSS you are actually using.

Prettier plugin for Tailwind CSS

One tiny annoyance with Tailwind is that there's no class sorting. This will result in less readable and consistent code. But there's a solution for this: Prettier plugin for Tailwind CSS. It auto sorts the tailwind classes upon formatting the file.

Prettier

Prettier is an opinionated code formatter. It comes with with a default config, but also allows you to create a configuration file where the formatting can be customized.

By using Prettier we will have a consistent coding style throughout the codebase.

Luxon

Luxon is a library for dealing with dates and times in JavaScript. We primarily used this for the math part of dealing with dates, for example: adding a month to date, keeping in mind all of the extra stuff that dates bring with them like leap years for example. This library makes that way easier.

Svelte-i18n

Elision request that we use translation keys throughout our code since this is an important part of the way they (and many others) build their frontends. Svelte-i18n is a great library to help us with this.

Zod

Zod is the validation library we use for our forms. It needs you to declare a schema that contains the fields of the form and the validation rules that go with it. It prides itself on being TypeScript-first so the typing for this library is excellent.

Next.js vs SvelteKit

Within Elision Next.js is the more used framework. However, they're considering a switch to SvelteKit as this does seem to provide some nice QoL features. This document will thus explore some of the differences between SvelteKit and Next.js.

What is Next.js

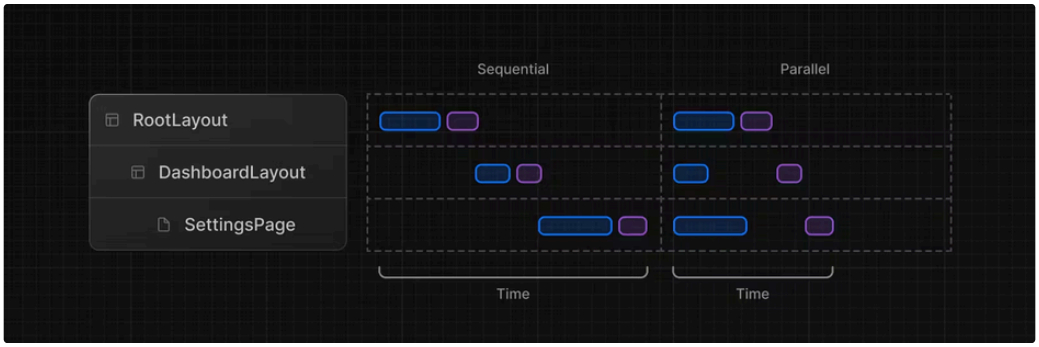
According to their official site Next.js is *the* react framework for the web, used by some of the world's largest companies. Next.js recently released their 14 version which stabilized some of their experimental features. In this document SvelteKit 2 with Svelte 5 will be compared with Next.js version 14.

SSR

Both frameworks support SSR, but implement this a bit differently. In Next.js SSR happens when using React Server Components (RSC), which is the default. Just like in SvelteKit there's multiple ways to render a page. there is static rendering or dynamic rendering (or a mix of both). Static rendering builds the page at build time. This is useful when a route has data that is not personalized to the user and can be known at build time, such as a static blog post or a product page. With Dynamic Rendering, routes are rendered for each user at **request time**, for example for a profile page. The difference however comes to light when comparing how you write the server code. In SvelteKit this is done in a `+page.server.svelte` file, in the `load` function. In Next.js this is done in a regular `.tsx` or `.jsx` file using a `fetch`.

Streaming

Streaming is the process of loading in data in a non-blocking way. For example on Amazon you might see the products at the bottom of the page load in faster than those a row above. This results in a better user experience because now the user doesn't have to wait for the data to load in sequentially.



Next.js docs: representation of streaming

Both Next.js and SvelteKit have the capability to handle this, but do it a little different. In Next.js you'd use the `<Suspense>` tag to wrap the component that needs the data. In SvelteKit you'd return promises from the `load` function in `+page.server.ts`. And then use an `{#await ...}` directive where needed.

Caching

Next.js goes all in on caching in its 14 version. For example: whenever a `fetch` is performed in a server component, the result gets cached. A disadvantage of Next.js caching is the way it's communicated to developers. It's more like magic to us. They could do a better job of conveying this.

How and what SvelteKit caches isn't really made very clear in their documentation, so we can't speak too much on that.

Server actions vs Form actions

Both of these boil down to the same thing: code that gets called by the client but runs on the server. How this code gets called though is a bit different. In both SvelteKit and Next.js these can be called through forms, by means of the `action` attribute. In SvelteKit it's constrained to this. In Next.js however they can also be called through event handlers and `useEffect`. In SvelteKit, if you want to call some server code outside of a form action you can use an API call, or do a workaround.

Developer experience

The way code is written in Svelte vs React may look kind of similar on the surface, but there are some differences. In Svelte you write your JS on top of your HTML. In React this is the other way around. The first looks and feels more logical. Svelte has templating which makes the HTML code more readable. Doing early returns with HTML generally doesn't feel to great and can get confusing if the code base grows. So you're often left with searching for a different way of doing things. In general it does feel like Svelte was made with modern web development in mind (html superset, efficient compiler) while React feels like it's trying to catch up. Even though it's userbase is much, much larger.

Ecosystem

React was released in 2013, Svelte in 2016. React has gained way more recognition than Svelte did. Maybe this is because of Facebook, a company that Svelte didn't have. React's introduction of the virtual DOM also helped them greatly.

Because it got more attention its ecosystem also got larger. There's lots of really nice packages for react that Svelte just doesn't have. To be fair some of the more important packages like the state management ones (jotai, redux, …) are already built-in to Svelte by means of universal reactivity.

Components

In NextJS (React) components are functions that return some JSX and thus can be placed anywhere in the file. If there is a `<div>` with some styling that's being used in a component and only that component, it would be better to extract it into its own component. It depends on the developer but most would just place that component in the same file (depending on size). This is a privilege not seen in Svelte. In Svelte every components is its own file. This caused us to keep smaller repetitive code inside of the same file as to not bloat our project structure with 10-liner components.

Documentation

Since React and NextJS are so much more used than Svelte and SvelteKit there are much more resources to be found for helping out if there is an issue. The lack of (good) documentation for Svelte is kind of a pain. It does have a passionate community, but if the documentation is no good then new developers will shy away from it pretty quickly.

CI/CD

Throughout the last 2 sprints of the project we had meetings with our mentors in order to discuss how we could proceed with deploying the application so it could be accessed outside of our laptops. Both our backend and frontend's code is located on Bitbucket. Elision had no build minutes on Bitbucket pipelines, but could provide some on Azure Pipelines. So our code remains on Bitbucket but the CD is handled by Azure pipelines. Seppe handled all of the deployment and setting up the CI/CD pipelines.

Frontend

Where

The frontend is deployed on Vercel, a PaaS that takes care of a lot of the annoyances of deployment.

Pipeline

The frontend pipeline is pretty simple. All it does is build the application using `pnpm build` and then pushes it to Vercel as a production deployment if the source branch is master.

```yaml
# Builds the application for any push/PR opened for master/dev.
# Deploys to Vercel only if branch is dev or master.

trigger:
- master
- dev

pr:
- master
- dev

pool:
  vmImage: ubuntu-22.04

variables:
  pnpm_config_cache: $(Pipeline.Workspace)/.pnpm-store
  isSourceMaster: $[eq(variables['Build.SourceBranch'], 'refs/heads/master')]

stages:
- stage: Build
  jobs:
    - job: BuildJob
      steps:
        - task: Cache@2
          inputs:
            key: 'pnpm | "$(Agent.OS)" | pnpm-lock.yaml'
            path: $(pnpm_config_cache)
          displayName: Cache pnpm

        - script: |
            corepack enable
            corepack prepare pnpm@latest-9 --activate
            pnpm config set store-dir $(pnpm_config_cache)
          displayName: "Setup pnpm"

```

```
        - script: |
            pnpm install
          displayName: "pnpm install"

        - script: |
            pnpm run build
          displayName: "pnpm build"
          env:
            API_BASE_URL: $(API_BASE_URL)
            STRAVA_CLIENT_ID: $(STRAVA_CLIENT_ID)
            STRAVA_CLIENT_SECRET: $(STRAVA_CLIENT_SECRET)

- stage: Deploy
  dependsOn: Build
  condition: eq(variables.isSourceMaster, true)
  jobs:
  - job: DeployJob
    steps:
      - task: vercel-deployment-task@1
        inputs:
          vercelProjectId: 'myProjectId'
          vercelOrgId: 'myOrgId'
          vercelToken: '$(VERCEL_TOKEN)'
          vercelCWD: '$(System.DefaultWorkingDirectory)'
          production: true

```

1. First of the `trigger` and `pr` are set. This pipeline will trigger whenever something is committed onto `master` or `dev` and also run whenever a new PR that targets either of them is opened or updated.
2. Next the agent is set. This pipeline runs on a Microsoft hosted agent, an `ubuntu-22.04` image.
3. Then some variables are declared, like the pnpm config cache and whether or not the source is master.
4. Next up is the first stage, build. It first does some pnpm setup and then installs all dependencies and tries to build the project. The environment variables come from the Azure pipeline variables itself.
5. Last up is the Deployment to Vercel. This depends on Build, meaning that has to run first and only runs if the earlier defined `isSourceMaster` variable equates to `true`. So it only runs whenever a commit to master is made, or in other words whenever we merge the `dev` branch into `master`. The deployment itself is handled by a task made by Vercel. This will deploy as a production build meaning our elicomp.vercel.app URL will then point to it.

Database

Where

Our database runs on a Debian-based virtual machine on Azure. Once the VM was up and running SSH can be used to access its CLI and begin setting it up. Firstly Seppe downloaded docker for Debian and then spun up a lightweight alpine PostgreSQL v16.3 container. Except for a custom password no additional environment variables were passed along to the container. After this a created a `prod` database was created and thus the database was set up. In order to connect the backend and database Seppe thought it would be a good idea to only expose the backend to the internet and keep it and the database in a virtual network so they could communicate.

Backend

Where

The backend runs in an Azure App Service. App service is an Azure service used to quickly deploy apps. It provides plenty of other advantages like advanced security, high availability, automating scaling, etc. It's basically a managed layer that takes away some control but should make maintaining easier. In hindsight this might have been a bit overkill for what we have, but we wanted to emulate an actual deployment.

Problems

Compared to the easy deployment of the frontend, this was in an whole other ball park. As mentioned earlier we had plenty of meetings discussing where to deploy to. Copying all our files over to simple Linux server came up too, but Seppe wanted to use services like AWS or Azure since that's what is being used all over the place now. There were however many, many issues with this.

Building most of the pipeline was easy. Just copy most of it over from the frontend and change the build step to use a Maven build task instead. This was done pretty quickly. The tricky part was the actual deployment to Azure. Due to restrictions placed on Seppe's Azure Elision account he couldn't create an application in Microsoft Entra (old: Microsoft Active Directory). This took a pretty long time to figure out and see if there was any fix (there was not). Seppe figured the best course of action would be to just use his student account then since students are entitled to some student credits on Azure. Seppe created all of the resources again and tried to deploy to Azure (student account) via Azure pipelines (Elision account → build minutes), but no luck. The school requires us to enable 2 factor authentication for security reasons and they are Microsoft partner. Because of this we can't use a simple email + password authentication option when attempting to authenticate before deploying to Azure. Seppe had to resort to service principal + certificate based authentication instead. After creating a service principal and certificate he tried again, but to no avail. He had passed along a wrong application secret. In order to find the correct application secret though, it seemed Seppe had to deal with Microsoft Entra again. The school had also disabled our access to this.

So then with seemingly only one option left (without giving Microsoft our credit card), Seppe created his own Azure Pipelines account and tried it that way. No luck. Even though we would not use parallel jobs (running multiple pipeline jobs at the same time) Microsoft required us to have them. For this we had to fill in a form that could take up to 3 business days to get an answer back on. Seppe tried to use his own machine to run the pipeline on, see self-hosted agent. But this was no big success and he wasn't really a fan of this. Seppe had a meeting with our mentors again to talk about some other options like AWS.

After laying this to rest for a bit Seppe suddenly got an email from Microsoft notifying him we got approved for running a singular parallel job. This meant we could get started now. From here on out Seppe unlinked the webhooks between the Azure Pipelines Elision account and our Bitbucket repos so we wouldn't use any unneeded build minutes there. Then he continued with trying to deploy our backend to Azure, but again something went wrong. Not any connection issues this time. About 15 minutes after he encountered the issue, Seppe had a planned meeting to go over the pipelines/deployment issues, which were now fixed. In this meeting was a deployment specialist from a sister company. He hadn't worked with Azure much so couldn't really help much. He did see that the issue Seppe was currently having was that our backend just wasn't running. Seppe then remembered that you could provide an optional startup command to the App Service. After trying this with our mentor who was very knowledgeable on Java and the deployment specialist, Seppe still couldn't get it working and they suggested to maybe start looking into AWS.

Once the meeting was done Seppe continued trying to get this to work because he got this far. It turned out, in the startup command, the path to the `.jar` file was simply wrong. After fixing this, everything worked perfectly, although a bit slow.

Pipeline

```
# Builds the java application and runs its tests.
# Deploys to either prod or dev if branch is either master or dev, and is not a PR.

trigger:
- master
- dev
```

```yaml

# Only PRs that target master and dev
pr:
- master
- dev

pool:
  vmImage: ubuntu-22.04

variables:
  isSourceMaster: $[eq(variables['Build.SourceBranch'], 'refs/heads/master')]

stages:
- stage: Build
  jobs:
  - job: BuildJob
    steps:
      - task: Maven@4
        inputs:
          mavenPomFile: 'pom.xml'
          publishJUnitResults: false
          javaHomeOption: 'Path'
          jdkDirectory: '$(JAVA_HOME_21_X64)'
          mavenVersionOption: 'Default'
          mavenAuthenticateFeed: false
          effectivePomSkip: false
          sonarQubeRunAnalysis: false
      - task: CopyFiles@2
        inputs:
          Contents: '**'
          TargetFolder: '$(build.artifactstagingdirectory)'
      - task: PublishBuildArtifacts@1
        inputs:
          PathtoPublish: '$(Build.ArtifactStagingDirectory)'
          ArtifactName: 'drop'
          publishLocation: 'Container'

- stage: Deploy
  dependsOn: Build
  condition: eq(variables.isSourceMaster, true)
  jobs:
  - job: DeployJob
    steps:
      - bash: |
          echo $(System.DefaultWorkingDirectory)
      - task: DownloadBuildArtifacts@1  # Download artifact from previous stage
        inputs:
          artifactName: 'drop'
          downloadPath: '$(System.DefaultWorkingDirectory)'
      - bash: |
          basename $(System.DefaultWorkingDirectory)/**/*.jar
          echo "##vso[task.setvariable variable=ArtifactBaseName;]$(basename $(System.DefaultWorkingDirectory)/*
      - task: AzureRmWebAppDeployment@4
        inputs:
          ConnectionType: 'AzureRM'
          azureSubscription: 'mySubscription'
          appType: 'webAppLinux'
          WebAppName: 'elicomp3'
```

```
          packageForLinux: '$(System.DefaultWorkingDirectory)/**/*.jar'
          RuntimeStack: 'TOMCAT|10.1-java21'
          StartupCommand: 'java -DJWT_SECRET_KEY=$(JWT_SECRET_KEY) -DDB_HOST=$(DB_HOST) -DDB_NAME=$(DB_NAME) -DD
        displayName: Send .jar to Azure app service

```

1. First of the `trigger` and `pr` are set. This pipeline will trigger whenever something is committed onto `master` or `dev` and also run whenever a new PR that targets either of them is opened or updated.
2. Next the agent is set. This pipeline runs on a Microsoft hosted agent, an `ubuntu-22.04` image.
3. Next up a variable is declared to determine whether or not the source is master.
4. Next up is the first stage, build. It runs the maven task and builds our java application to check for any build issues, but it also runs all of our tests. Seppe did not take coverage into account due to time constraint and project focus. If this succeeds it published an artifact of our application.
5. The deploy stage only runs once the build stage completes and only if the source branch is master. It first downloads the artifact and then sends it over to Azure. The startup command can also be seen in the `AzureRmWebAppDeployment` task. We pass along some system properties using `-D`.