4 - Load and display data

In the previous chapters, we have learned how to set up a SvelteKit application and how to connect it to a database. We have also seen how to create Drizzle ORM models, and how to execute basic queries using the Drizzle ORM.
In this chapter, we will learn how to use these queries to load the data from the database to the SvelteKit app, and display it using various visualization techniques.

Chapter overview

Learning goals

By the end of this chapter, you will be able to:

- Understand and use the load function in SvelteKit to preload data for your pages.
- Distinguish between universal and server-only load functions.
- Query data from a database using Drizzle ORM and safely pass it into your frontend.
- Create and use Data Access Objects (DAOs) to encapsulate and modularize query logic.
- Use route parameters and URL queries to implement dynamic and responsive data loading.
- Handle errors gracefully using SvelteKit's error handling mechanisms.
- Display data in various formats:
  - Basic component-based rendering.
  - Reusable table components with optional pagination.
  - Dynamic charts using Chart.js for visual insights.
- Extract utility functions to promote reuse and clarity in your codebase.

Learning resources

- SvelteKit Documentation
- Drizzle ORM Documentation
- Chart.js Documentation
- Skeleton.dev UI Components
- Understanding DAOs (Data Access Objects)

Data loading with SvelteKit

Loading page data

Let's say we want to show a list of users from our database in a SvelteKit application.
To do this, we would need a UI element that displays the list of users, and load the data from the database into that UI element before rendering it.

The load function allows us to do just that. It is a special function that lives in a sibling `+page.ts` or `+page.server.ts` file, next to a `+page.svelte` file.

The `load` function is called before the page is rendered, and it's return value is passed to the page as a `data` prop.
We can use this function to load data from a database, API, or any other source, and then pass it to the page for rendering.
So in our case, we will use the `load` function to query our database for the list of users, and then pass that data to the page, which will then use the data to render the list of users.

Universal loading vs server-side loading

It's important to note that there are two types of load functions in SvelteKit: **universal** and **server-side**.
Universal load functions are defined in `+page.ts` files, while server-side load functions are defined in `+page.server.ts` files. The main difference between the two is that server-side load functions run only on the server, while universal load functions can run on both the server and the client.

Without getting too deep into the details, here's a quick description of when to use each type from the SvelteKit documentation:

> Server load functions are convenient when you need to access data directly from a database or filesystem, or need to use private environment variables.
> Universal load functions are useful when you need to fetch data from an external API and don’t need private credentials, since SvelteKit can get the data directly from the API rather than going via your server. They are also useful when you need to return something that can’t be serialized, such as a Svelte component constructor.
> In rare cases, you might need to use both together — for example, you might need to return an instance of a custom class that was initialised with data from your server. When using both, the server load return value is not passed directly to the page, but to the universal load function (as the data property)

As we will be loading data from a database, we will use the server-side load function in this chapter.

Loading static data

Before we start querying our database, let's just load some static data to see how the `load` function works.

We will create a new `+page.server.ts` file in the `src/routes` directory, and add the following code:

```
// /src/routes/+page.server.ts

import type { PageServerLoad } from "./$types";

export const load: PageServerLoad = async () => {
    const myData = {
        name: "John Doe",
        age: 30,
        display: true
    };
    return { myData };
}
```

We import the `PageServerLoad` type from the `$types` module, which is a generated module. This ensures that our `load` function passes the correct type on to the `data` prop that the page will receive.

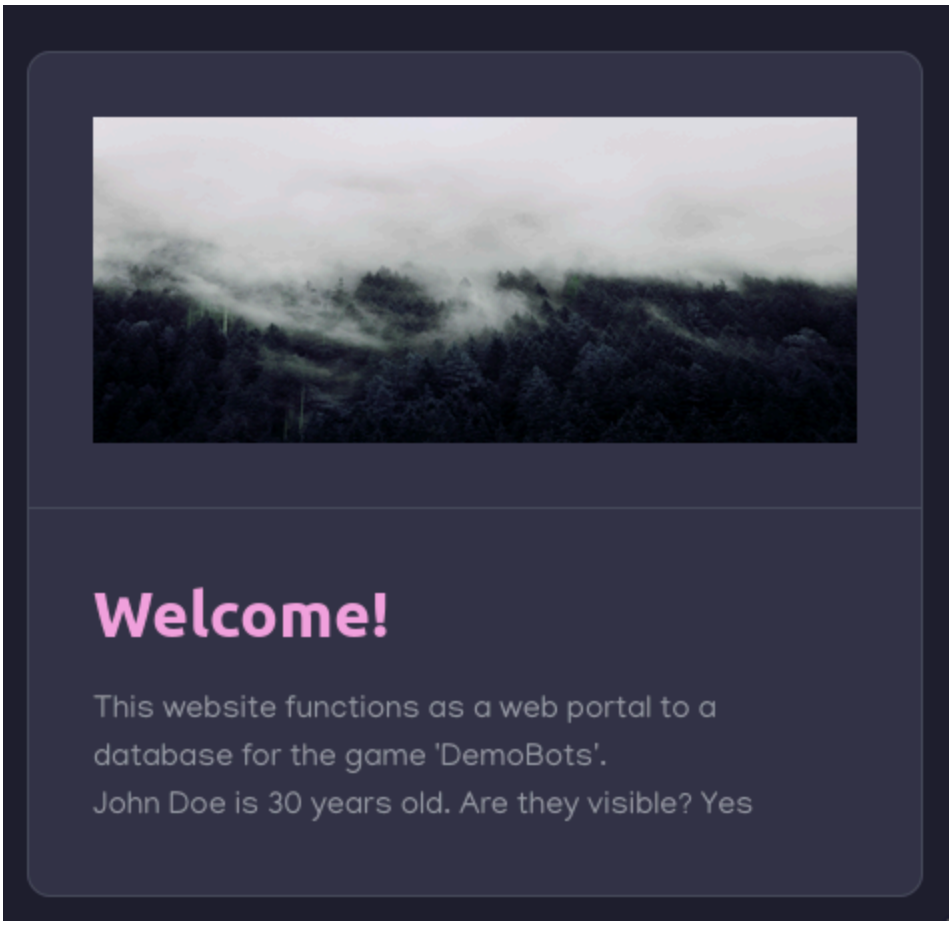
We then define the `load` function itself, which returns an object titled `myData`, with some static data.

Now, we can use this data in our `+page.svelte` file at the same level as the `+page.server.ts` file (`src/routes/+page.svelte`).
Let's quickly add some code to display this data from `myData`:

```
<!-- /src/routes/+page.svelte -->

<script lang="ts">
    ...
    let { data } = $props();
</script>

<div
    class="..."
>
    <Card footerBase="...">
        {#snippet header()}
            ...
        {/snippet}
```

```
        {#snippet article()}
            ...
            <p class="opacity-60">
                ...
                {data.myData.name} is {data.myData.age} years old.
                Are they visible? {data.myData.display ? 'Yes' : 'No'}
            </p>
        {/snippet}
    </Card>
</div>
```

Et voilà! When we visit the page, we should now see that static data displayed on the page:



Let's remove the code we just added in the `+page.svelte` file, and replace the `load` function in the `+page.server.ts` file with some code that queries our database for a list of users.

Loading data from the database

To load data from the database, we will use the Drizzle ORM that we set up in the previous chapter.

First, to retain type safety between our database schema and the data we show in the SvelteKit app, we can infer the types for the data directly from the tables we defined in our Drizzle ORM

schema.
To export the types for our Schema, we'll add the following code to (the very bottom of) our `src/lib/db/schema.ts` file:

```
export type User = typeof users.$inferSelect;
export type UserProfile = typeof userProfiles.$inferSelect;
export type Session = typeof sessions.$inferSelect;
export type Score = typeof scores.$inferSelect;
export type Level = typeof levels.$inferSelect;
```

Next, let's rework the `+page.server.ts` file to query the database for a list of users.

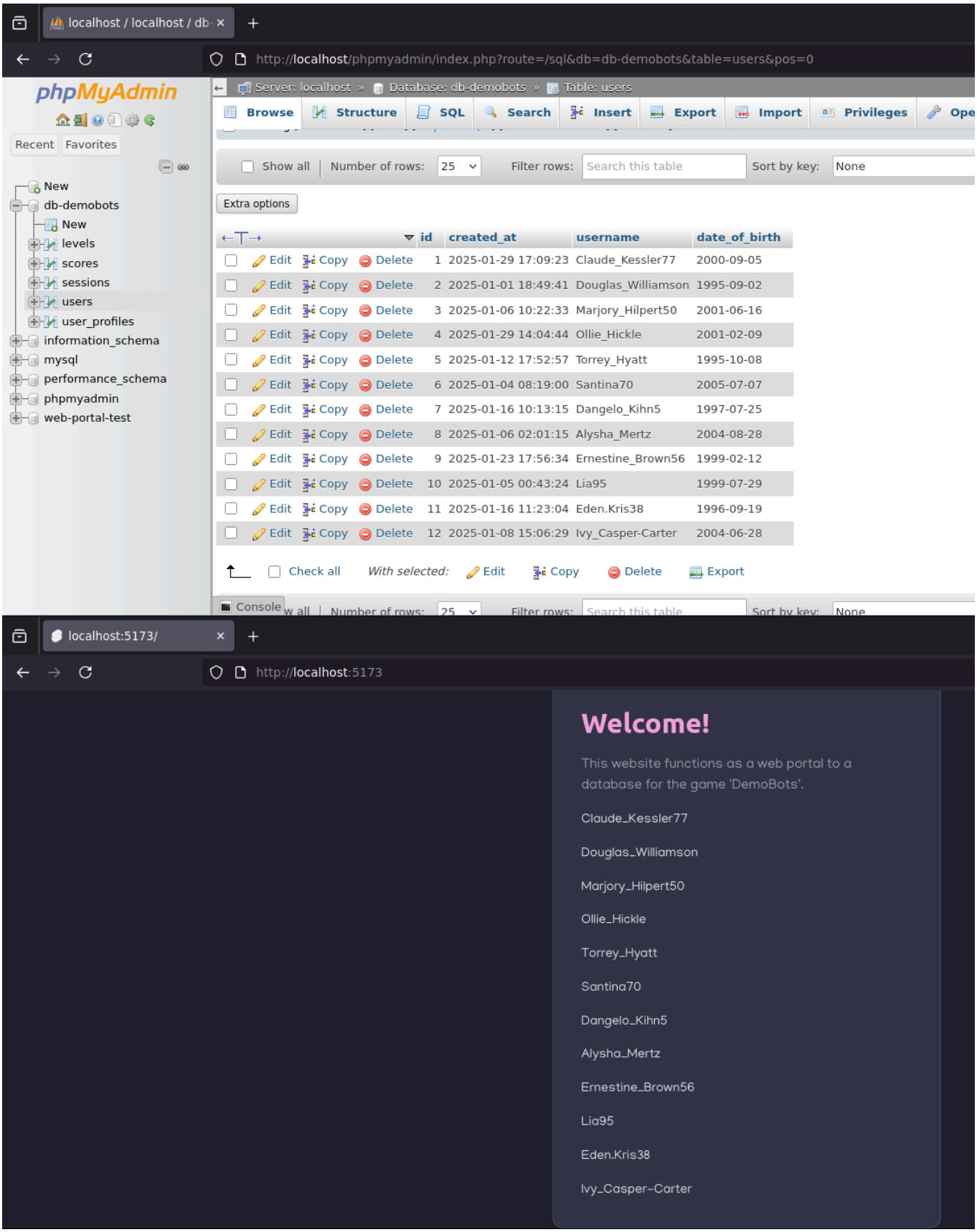
```
// /src/routes/+page.server.ts

import type { PageServerLoad } from "./$types";

// We import the db instance and the schema we defined in the previous
chapter
import { db } from "$lib/server/db";
import * as schema from "$lib/server/db/schema";
// We import the newly inferred User type
import type { User } from "$lib/server/db/schema";

export const load: PageServerLoad = async () => {
    // Here we perform a basic select query to get all users from the
database
    const users: User[] = await db.select().from(schema.users);
    // And finally, we pass the list of users to the page's data prop
    return {
        users
    }
}
```

And then update the `+page.svelte` file to display that list of users.
We'll make use of the `$derived` rune to ensure that the list of users is reactive, meaning that if the data changes, the UI will automatically update.
(Though in our example, the data won't change reactively. But this is generally a good practice to follow.)

```
<!-- /src/routes/+page.svelte -->

<script lang="ts">
    ...
```

```
    // We import the User type to ensure type safety
    import type { User } from '$lib/server/db/schema';

    let { data } = $props();

    // We use the $derived rune to create a reactive list of users
    let users: User[] = $derived(data.users);
</script>

<div
    class="..."
>
    <Card footerBase="...">
        ...
        {#snippet article()}
            ...
            <!-- We display the list of users -->
            {#each users as user (user.id)}
                <p class="text-sm">
                    {user.username}
                </p>
            {/each}
        {/snippet}
    </Card>
</div>
```

Let's take a look at the `/` route now, and confirm that we are displaying the data from the users in our database:



Awesome! We now know how to load data from the database and display it in our SvelteKit application.

Adding a DAO

In the previous section, we loaded the data directly in the `+page.server.ts` file.
While this works, it can quickly become hard to manage the data loading logic as our application grows.

To make our code more modular and maintainable, we can create a Data Access Object (DAO) that encapsulates the logic for loading data from the database. It's a layer that sits between our database and our application, allowing us to abstract away the details of how we access the data.
This simplifies the code in our `+page.server.ts` file to simply calling the DAO methods to get the data we need.

Let's create a new file called `DAO.ts` in the `src/lib/server/dao` directory, and add the following code:

```
import { db } from '$lib/server/db';
import * as schema from '$lib/server/db/schema';
import type { MySql2Database } from 'drizzle-orm/mysql2';

export abstract class DAO {
    protected static readonly db: MySql2Database<typeof schema> = db;
}
```

This will form an abstract base class for our `DAO` classes.
Next, we will create a `UserDAO` class that extends this `DAO` class and provides methods for loading users from the database.

```
// /src/lib/server/dao/UserDAO.ts

import { DAO } from "$lib/server/dao/DAO";
import type { User } from "$lib/server/db/schema";
import { users } from "$lib/server/db/schema";

export class UserDAO extends DAO {
    static async getAllUsers(): Promise<User[]> {
        return DAO.db.select().from(users);
    }
}
```

Now, let's use this `UserDAO` class in our `+page.server.ts` file to load the users from the database:

```
// /src/routes/+page.server.ts
```

```
import type { PageServerLoad } from "./$types";

import type { User } from "$lib/server/db/schema";
import { UserDAO } from "$lib/server/dao/UserDAO";

export const load: PageServerLoad = async () => {
    const users: User[] = await UserDAO.getAllUsers();
    return {
        users
    }
}
```

There, that looks much cleaner! Especially once we start adding more multiple and more advanced queries to our DAO, this will help us keep our code organized and maintainable.

This is neat and all, but we don't really need to show a list of users on our home page. We know how to do that now, but for now let's undo the changes we made to the `+page.svelte` file, and remove the `+page.server.ts` file as well.

Next, we'll be looking at implementing a more complex query, and display the results in a component.

Querying with the where clause: Selecting a specific user

Let's say that we want to display some user data, but only for a specific user, identified by their ID.
We *could* use the `UserDAO` class we created earlier to get all users, and then filter the list to find the user we want. But, that would put unnecessary load on the database.
We could instead just query the database for that specific user.

Let's look at how we can do that. Let's add a method to our `UserDAO` class `findUserById` that takes a user ID as an argument and returns the user with that ID from the database:

```
// /src/lib/server/dao/UserDAO.ts

...
import { eq } from 'drizzle-orm';

export class UserDAO extends DAO {
    ...

    static async getUserById(id: number): Promise<User | undefined> {
        return DAO.db.query.users.findFirst({
            where: eq(users.id, id)
```

```
        });
    }
}
```

> ℹ️ Note that we could also write this query using the `select` method: `DAO.db.select().from(users).where(eq(users.id, id)).then(rows => rows[0]);`. The `findFirst` method is just a convenience method.

Now, let's make a new route at `/src/routes/dashboard/user/[id]/` (note the square brackets around `id`, we will explain those in a moment) and create a `+page.server.ts` file in that directory.

```ts
// /src/routes/dashboard/user/[id]/+page.server.ts

import type { PageServerLoad } from "./$types";

import type { User } from "$lib/server/db/schema";
import { UserDAO } from "$lib/server/dao/UserDAO";

export const load: PageServerLoad = async () => {
    const id = 3;
    const user: User | undefined = await UserDAO.findUserById(id);

    return {
        user
    }
}
```

And the corresponding `+page.svelte` file:

```svelte
<!-- /src/routes/dashboard/user/[id]/+page.svelte -->

<script lang="ts">
    import type { User } from '$lib/server/db/schema';

    let { data } = $props();

    let user: User = $derived(data.user);

</script>

<p>
    User: {user.username}
</p>
```

Now, if we visit the `/dashboard/user/3` route, we should see the username of the user with ID 3 displayed on the page.

But if we visit `/dashboard/user/4`, we will still see the same user displayed, because we hardcoded the ID in the `+page.server.ts` file.
This is where the `[id]` part of the route comes in. It is a routing feature of SvelteKit, and it allows us to create dynamic routes that can accept parameters.

We can access the `id` parameter in the `load` function by passing the `params` parameter to the `load` function.

```
// /src/routes/dashboard/user/[id]/+page.server.ts

...

export const load: PageServerLoad = async ({ params }) => {
    const id: number = parseInt(params.id);
    const user: User | undefined = await UserDAO.findUserById(id);

    return {
        user
    }
}
```

Now, if we visit `/dashboard/user/4`, we should see the username of the user with ID 4 displayed on the page instead.

Let's do the same for the `/dashboard/session/[id]` route, which will display the details of a specific session. We will get into less detail here, as the code is mostly repetitive of what we did for the user route.

First, we'll create the `SessionDAO`:

```
// /src/lib/server/dao/SessionDao.ts

import { DAO } from '$lib/server/dao/DAO';
import { type Session, sessions } from '$lib/server/db/schema';
import { eq } from 'drizzle-orm';

export class SessionDAO extends DAO {
    static async getSessionById (id: number): Promise<Session | undefined> {
        return await DAO.db.query.sessions.findFirst({
            where: eq(sessions.id, id),
        });
```

```
    }
}
```

Next, we'll create the `+page.server.ts` file in the `src/routes/dashboard/session/[id]` directory:

```
// /src/routes/dashboard/session/[id]/+page.server.ts

import type { PageServerLoad } from './$types';

import type { Session } from '$lib/server/db/schema';
import { SessionDAO } from '$lib/server/dao/SessionDAO';

export const load: PageServerLoad = async ({ params }) => {
    const id: number = parseInt(params.id);

    const session: Session | undefined = await
SessionDAO.getSessionById(id);

    return {
        session
    };
};
```

And finally, create the corresponding `+page.svelte` file:

```
<!-- /src/routes/dashboard/session/[id]/+page.svelte -->

<script lang="ts">
    import type { Session } from '$lib/server/db/schema';

    let { data } = $props();

    let session: Session = $derived(data.session);
</script>

<p>
    Session: {session.id}
</p>
```

Now we can visit both user and session pages in the dashboard. We'll build them out later to display more information, but for now we can confirm that we can query a specific user or session by ID.

Querying joined tables: Selecting the top 10 users

Now that we know how to query a single user by ID, let's look at how we can query the top 10 users.

We'll define the top 10 users as the users who have achieved the highest scores, across all sessions and levels.
So, for example, though one of our users may have a score of 100 and 95 in two different sessions and levels, we will only count the highest score of 100 for that user.
Our next top user may have a maximum score of 95, and so on, until we have the top 10 users.
If two users have the same maximum score, we won't order them (though we could, e.g. by date).

Let's create a new `ScoreDAO` that will handle the logic for querying scores from the database.

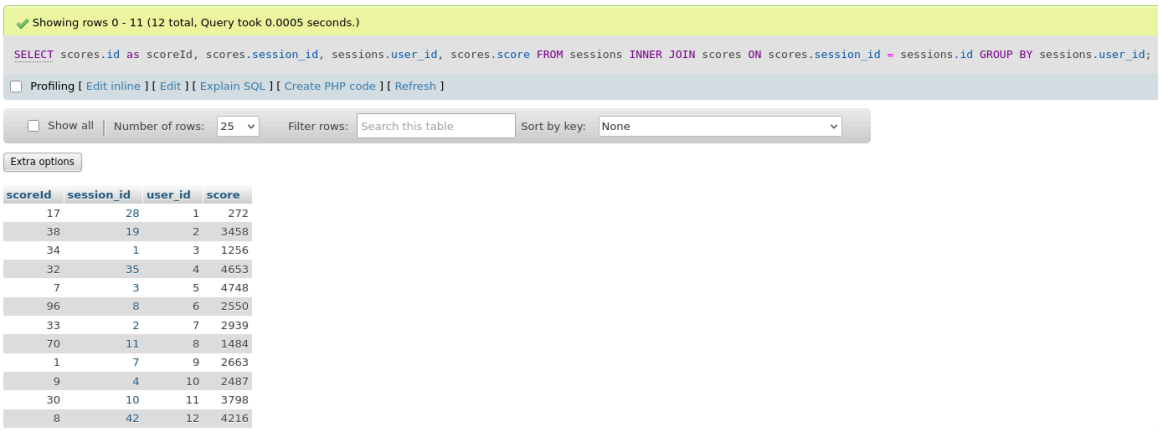
```
// /src/lib/server/dao/ScoreDAO.ts

import { DAO } from '$lib/server/dao/DAO';
import {
    users,
    scores,
    sessions,
    type User,
    type Score,
    type Session
} from '$lib/server/db/schema';
import { eq, and, desc, max } from 'drizzle-orm';

export class ScoreDAO extends DAO {
    static async getTopScorers(limit: number): Promise<TopScorer[]> {
        // First, find the maximum score for each user
        const maxScoreSubquery = DAO.db
            // We want to find the maximum score for each user,
            // across all their sessions.
            .select({
                // We want to select the userId that achieved the max score
...
                userId: sessions.userId,
                // ... and the maximum of all scores for that user.
                maxScore: max(scores.score).as('max_score')
            })
            // We start with the sessions (the bridge between users and
scores) ...
            .from(sessions)
            // ... and join with the scores table to get the scores for each
```

```
session ...
            .innerJoin(scores, eq(scores.sessionId, sessions.id))
            // ... then group by userId to return all scores per user ...
            .groupBy(sessions.userId)
            // ... and finally alias this subquery so it can be referenced
later.
            .as('max_scores');

        // Then find the score records that match these maximums
        return await DAO.db
            // Select both user and score information ...
            .select({ user: users, score: scores, session: sessions })
            // ... from the users table...
            .from(users)
            // ... joined with the maximum scores subquery ...
            .innerJoin(maxScoreSubquery, eq(maxScoreSubquery.userId,
users.id))
            // ... joined with the sessions that match the users id ...
            .innerJoin(sessions, eq(sessions.userId, users.id))
            // ... and the score that matches both the relevant sessionId
and the maximum score ...
            .innerJoin(
                scores,
                and(eq(scores.sessionId, sessions.id), eq(scores.score,
maxScoreSubquery.maxScore))
            )
            // ... then order the results by score in descending order ...
            .orderBy(desc(scores.score))
            // ... and limit the results to the top 10 scorers.
            .limit(limit);
    }
}

// Stores the relevant User, Score, and Session data for a top scorer
export interface TopScorer {
    user: User;
    score: Score;
    session: Session;
}
```

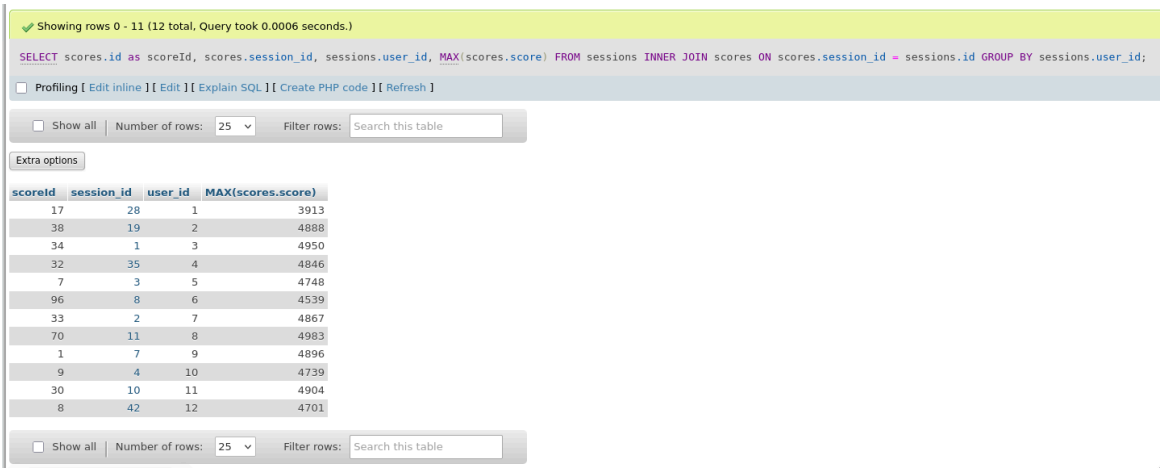
Key points to note here are:

- We first run a subquery to find the maximum score for each user, across all their sessions.
- We then join this subquery with the users, sessions, and scores tables to get the relevant data for each top scorer.

- We use the `max` function from Drizzle ORM to find the maximum score for each user (which executes the sql `MAX` function).
  - If we wouldn't, we would just get one of the scores for each user, which may not be the maximum score.
  - Inner joins like this can be a bit unintuitive:
    - A query only returns one value per row per column.
    - In the subquery, we have multiple scores per session
    - When we inner join the scores on the sessions table, and then group by userId, we get only one score value per row, arbitrarely chosen by position in the database.



    - But the `MAX` function ensures the value is the Maximum of all possible values that could fit that row:



  - We can see that the other columns (e.g. session id) don't change, but that's okay, because we're only after the user id and the MAX score in this subquery.
  - We will find the corresponding score, user, and session in the next query.
- We use the `as` method to alias the maximum score column, so we can reference it later in the query.
  - We don't actually need this alias for the drizzle syntax, but if we don't add it, drizzle can't create the relevant SQL, as we need to alias subqueries.

- We use the `and` function from Drizzle ORM to combine multiple conditions in the `innerJoin` method.
- We use the `desc` function from Drizzle ORM to order the results by score in descending order.
- We limit the results to the top 10 scorers using the `limit` method.
- We create an interface `TopScorer` to store the relevant data for each top scorer, which includes the user who achieved the score, the score record itself, and the session the score was achieved in.
    - We map the `users`, `scores`, and `sessions` tables to this type in the `select` method. (e.g. if we select `users` (imported from our schema) we select all columns from the `users` table, and the return data is stored in an object with the `users` key. Here, we rename that key to `user` for clarity, as the returned data concerns a single user, not a list of users.)

Now, let's use this `ScoreDAO` in our `+page.server.ts` file to load the top scorers from the database.
We'll display the top scorers in the dashboard overview, so at `/dashboard`.

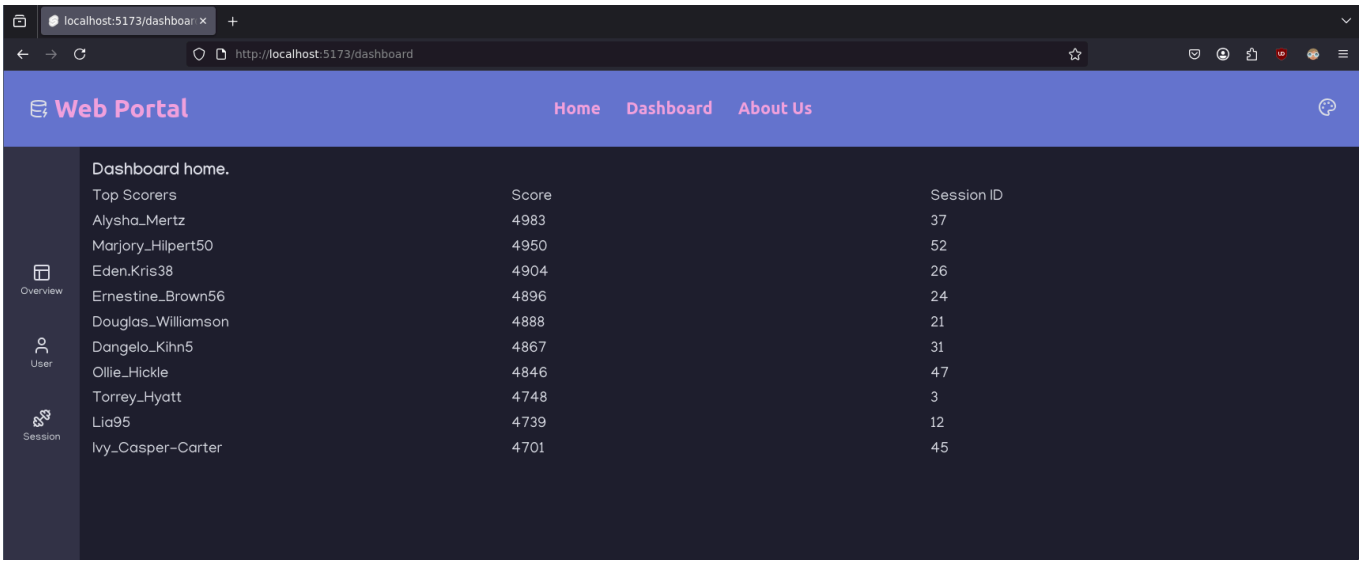
By now, we should know we should create a new `+page.server.ts` file in the `src/routes/dashboard` directory, and add the following code:

```
// /src/routes/dashboard/+page.server.ts

import type { PageServerLoad } from './$types';

import { ScoreDAO, type TopScorer } from '$lib/server/dao/ScoreDAO';

export const load: PageServerLoad = async () => {
    const limit: number = 10;

    return {
        topScorers
    };
};
```

```
const topScorers: TopScorer[] = await ScoreDAO.getTopScorers(limit);
```

Finally, let's create a rudimentary UI to display the top scorers in the `/routes/dashboard/+page.svelte` file.

```
<!-- /src/routes/dashboard/+page.svelte -->

<script lang="ts">
    import type { TopScorer } from '$lib/server/dao/ScoreDAO';

    let { data } = $props();

    let topScorers: TopScorer[] = $derived(data.topScorers);
</script>

<h2 class="text-xl font-bold">Dashboard home.</h2>

{#each topScorers as scorer (scorer.user.id)}
    <div class="flex items-center">
        <span class="text-lg">Username: {scorer.user.username}</span>
        <span class="text-lg">Score: {scorer.score.score}</span>
        <span class="text-lg">In session: {scorer.session.id}</span>
    </div>
{/each}
```



Awesome! That was a pretty advanced query, but we can see that Drizzle makes it quite easy to work on complex queries like this.

> ℹ️ If you want to, you can browse around in phpMyAdmin to confirm that these top scorers are indeed the top scorers in your database.

Responsive data loading

For our `/dashboard/user` page and the `/dashboard/session` page, we want users to be able to query for a specific user (by username) or session (by username or session ID).

We *could* create a form in which users can enter their query, and then submit that form to the server to load the data.
While this works, it is not as responsive as it can be: Let's try showing the results *as the user types their query*, without having to submit the form and wait for a new page to load.

To do this, we'll combine URL data and the data-sveltekit-keepfocus attribute.
How it works is that we will create a form with an input field. As the user types in the input field, we will update the URL with the query they entered.
The `load` function will then automatically run, and return the data from the database based on the query in the URL.
The `data-sveltekit-keepfocus` attribute will ensure that the input field remains focused, so the user can continue typing while querying the database.
This will make it feel like a responsive search, as the user will see the results update in real-time as they type.

First, let's expand the `UserDAO` class to include a method that finds users by their username, using the `like` operator to allow for partial matches:

```
// /src/lib/server/dao/UserDAO.ts
...
import { like } from 'drizzle-orm';

export class UserDAO extends DAO {
    ...

    static async findUsersLikeUsername(username: string): Promise<User[]> {
        return DAO.db.query.users.findMany({
            where: like(users.username, `%${username}%`)
        });
    }
}
```

Now we'll create a new file at `/src/routes/dashboard/user/+page.server.ts` to handle the loading based on the username query:

```
// /src/routes/dashboard/user/+page.server.ts

import type { User } from '$lib/server/db/schema';
import type { PageServerLoad } from './$types';
import { UserDAO } from '$lib/server/dao/UserDAO';

export const load: PageServerLoad = async ({ url }) => {
    // Here we get the username query parameter from the URL
```

```
    const username = url.searchParams.get('username');
    let users: User[] = [];
    if (username) {
        users = await UserDAO.findUsersLikeUsername(username);
    }
    return { users };
};
```

Next, we'll update the `/src/routes/dashboard/user/+page.svelte` file to include a form with an input field for the username query.

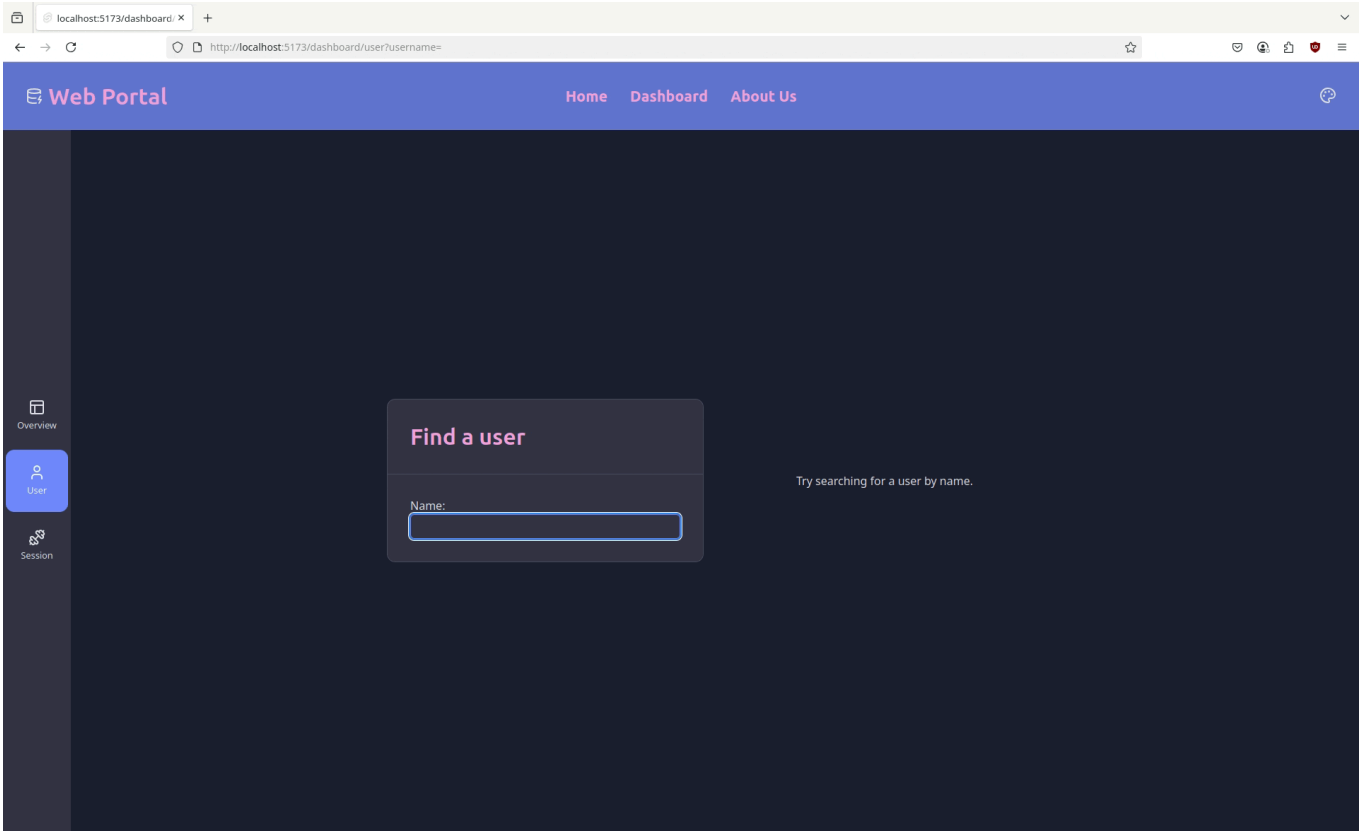
```
<!-- /src/routes/dashboard/user/+page.svelte -->

<script lang="ts">
    import Card from '$lib/ui/views/Card.svelte';
    import type { User } from '$lib/server/db/schema';

    let { data } = $props();
    let userResults: User[] | undefined = $derived(data.users);

    function searchForUsersByName(event: Event & { currentTarget:
EventTarget & HTMLInputElement }) {
        event.currentTarget.form?.requestSubmit();
    }
</script>

<div
    class="mx-auto my-24 grid grid-cols-1 items-center gap-4 sm:my-48 lg:my-
auto lg:grid-cols-[auto_1fr] lg:gap-8"
>
    <Card baseExtension="lg:min-w-md">
        {#snippet header()}
            <h1>Find a user</h1>
        {/snippet}
        {#snippet article()}
            <form data-sveltekit-keepfocus>
                <label for="username">Name: </label>
                <input class="input" type="text" name="username" oninput=
{searchForUsersByName} />
            </form>
        {/snippet}
    </Card>
    {#if userResults && userResults.length > 0}
        <Card baseExtension="lg:min-w-md">
            {#snippet header()}
```

```
                <h1>Found users</h1>
            {/snippet}
            {#snippet article()}
                <div class="flex max-h-64 flex-col overflow-y-scroll">
                    {#each userResults as user (user.id)}
                        <a href="/dashboard/user/{user.id}">{user.username}
</a>
                    {/each}
                </div>
            {/snippet}
        </Card>
    {:else}
        <div class="flex flex-col items-center justify-center lg:min-w-md">
            <p>Try searching for a user by name.</p>
        </div>
    {/if}
</div>
```

Key points to note here:

- We create a form with an input field for the username query.
- We use the `data-sveltekit-keepfocus` attribute on the form to ensure that the input field remains focused when the form is submitted.
- We use the `oninput` event on the input field to call the `searchForUsersByName` function, which submits the form when the user types in the input field.
- We show the list of users that are derived from the `data` prop, which is populated by the `load` function in the `+page.server.ts` file.
- If there are no found users, we show a message prompting the user to search for a user by name instead of the (empty) list of users.
- We make the usernames in the list clickable, so that the user can navigate to the user's detail page (`/dashboard/user/[id]`).

Now we should see the relevant users appear as we type:



Let's do the same for the sessions, so we can search for sessions by the session ID or the username of the user who created the session.

> ℹ️ Note that this does put more strain on the database, as it will query the database every time the user alters the input field. So this user friendliness comes at a cost. In a real, scalable application, you would likely want to implement some form of debouncing (so that the database is not queried on every keystroke, but rather after a short delay (e.g. 300ms) after the user stops typing), caching (so that the database is not queried for the same input multiple times), or pre-fetching all users (so that the database is queried only once, and the results are cached for subsequent queries).
> We'll expand the `SessionDAO` class, and add a method to find sessions by the session ID or the username of the user who created the session, much like we did for the users.

```
// /src/lib/server/dao/SessionDao.ts

...
import { type Session, type User, sessions, users } from
'$lib/server/db/schema';
import { eq, like } from 'drizzle-orm';

export class SessionDAO extends DAO {
    ...

    static async getSessionsLikeId(id: number): Promise<SessionWithUser[]> {
```

```
        const result = await DAO.db
            .select({ session: sessions, user: users })
            .from(sessions)
            .innerJoin(users, eq(sessions.userId, users.id))
            .where(like(sessions.id, `${id}%`));
        return result.map((row) => ({
            ...row.session,
            user: row.user
        }));
    }

    static async getSessionsLikeUserName(name: string):
Promise<SessionWithUser[]> {
        const result = await DAO.db
            .select({
                session: sessions,
                user: users
            })
            .from(sessions)
            .innerJoin(users, eq(sessions.userId, users.id))
            .where(like(users.username, `%${name}%`));
        return result.map((row) => ({
            ...row.session,
            user: row.user
        }));
    }
}

export type SessionWithUser = Session & {
    user: User;
};
```

And again, much like we did for the users, we'll create a new `+page.server.ts` file in the `src/routes/dashboard/session` directory, and make a `+page.svelte` file with a search form that allows users to search for sessions by ID or username.
We display the results in a rudimentary list of clickable links, and use the `data-sveltekit-keepfocus` attribute to keep the input field focused as the user types.
We're not going to go into too much detail here, as the code is very similar to what we did for the users: The only real 'new' thing here is that we now clear the other input field when the user types in one of the input fields, so we can discern between searching by ID or username.

```
// /src/routes/dashboard/session/+page.server.ts

import type { PageServerLoad } from './$types';
```

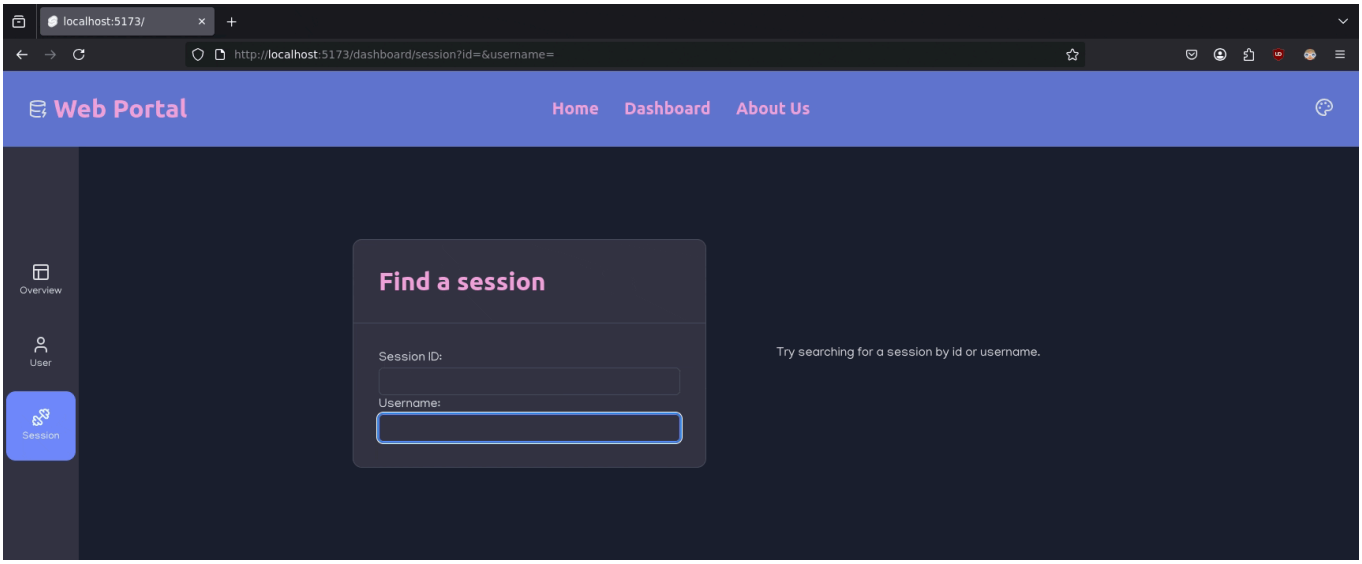
```
import { SessionDAO, type SessionWithUser } from
'$lib/server/dao/SessionDAO';

export const load: PageServerLoad = async ({ url }) => {
    let sessions: SessionWithUser[] = [];

    // Either query by username or by id
    const userNameParam = url.searchParams.get('username');
    if (userNameParam) {
        sessions = await SessionDAO.getSessionsLikeUserName(userNameParam);
        return { sessions };
    }

    const idParam = url.searchParams.get('id');
    const id = idParam ? parseInt(idParam) : null;
    if (id) {
        sessions = await SessionDAO.getSessionsLikeId(id);
    }
    return { sessions };
};
```

```
<!-- /src/routes/dashboard/session/+page.svelte -->

<script lang="ts">
    import Card from '$lib/ui/views/Card.svelte';
    import { type SessionWithUser } from '$lib/server/dao/SessionDAO';

    let { data } = $props();
    let sessionResults: SessionWithUser[] | undefined =
$derived(data.sessions);

    let idInput: HTMLInputElement, usernameInput: HTMLInputElement;

    function searchForSessionsById(event: Event & { currentTarget:
EventTarget & HTMLInputElement }) {
        usernameInput.value = '';
        event.currentTarget.form?.requestSubmit();
    }

    function searchForSessionsByUsername(
        event: Event & { currentTarget: EventTarget & HTMLInputElement }
    ) {
        idInput.value = '';
        event.currentTarget.form?.requestSubmit();
    }
```

```
</script>

<div class="lg: m-auto grid grid-cols-1 items-center gap-4 lg:grid-cols-
[auto_1fr] lg:gap-8">
    <Card baseExtension="lg:min-w-md">
        {#snippet header()}
            <h1>Find a session</h1>
        {/snippet}
        {#snippet article()}
            <form data-sveltekit-keepfocus>
                <label for="id">Session ID: </label>
                <input
                    class="input"
                    type="text"
                    name="id"
                    oninput={searchForSessionsById}
                    bind:this={idInput}
                />
                <label for="id">Username: </label>
                <input
                    class="input"
                    type="text"
                    name="username"
                    oninput={searchForSessionsByUsername}
                    bind:this={usernameInput}
                />
            </form>
        {/snippet}
    </Card>
    {#if sessionResults && sessionResults.length > 0}
        <Card baseExtension="lg:min-w-md">
            {#snippet header()}
                <h1>Found Sessions</h1>
            {/snippet}
            {#snippet article()}
                <div class="max-h-64 overflow-y-scroll">
                    {#each sessionResults as result (result.session.id)}
                        <div class="flex flex-col gap-2">
                            <a
href='/dashboard/session/{result.session.id}'>
                                <span>Session {result.session.id} by
{result.user.username}</span>
                            </a>
                        </div>
                    {/each}
                </div>
```

```
            {/snippet}
        </Card>
    {:else}
        <div class="flex flex-col items-center justify-center lg:min-w-md">
            <p>Try searching for a session by id or username.</p>
        </div>
    {/if}
</div>
```

Now we can search for sessions by either ID or username, and the results will be displayed in a list of clickable links that lead to the session detail page:



Error handling

The keen eyed among you may have noticed that we are not handling error cases yet, such as where the user with the given ID does not exist in the database.
Let's add some error handling to our `load` function to handle this case.

We can use Svelte's error function for this, and define a custom `+error.svelte` file (another SvelteKit routing feature) to display the error message.

```
// /src/routes/dashboard/user/[id]/+page.server.ts

...
import { error } from "@sveltejs/kit";

export const load: PageServerLoad = async ({ params }) => {
    const id: number = parseInt(params.id);

    const user: User | undefined = await UserDAO.findUserById(id);
```

```
    if (!user) {
        // This will just log the error to our console, so we're aware of
it.
        console.error(`User with ID ${id} not found. Showing 404.`);
        // This will throw a 404 error, which will be caught by the
+error.svelte file.
        throw error(404, `User with ID ${id} not found`);
    }

    return {
        user
    }
}
```

```
<!-- /src/routes/+error.svelte -->
<script lang="ts">
    import { page } from '$app/state';
</script>

<div class="text-center p-8">
<h1 class="text-3xl font-bold">Error {page.status}</h1>
    <p>{page.error.message}</p>
</div>
```

Notable is that the `+error.svelte` file inherits layouts depending on where it's placed.
For example, if we place a `+error.svelte` file at `src/routes` it will be used to display all errors, using the layout from the `src/routes/+layout.svelte` file.
If we then add another `+error.svelte` file at `src/routes/dashboard`, that file will be used to display all errors in the `/dashboard` route (and its subroutes).
So to not lose the sidebar navigation in the `dashboard` routes, let's extract an `Error` component, and use it in both `src/routes/+error.svelte` and `src/routes/dashboard/+error.svelte`.

```
<!-- /src/lib/ui/views/Error.svelte -->

<script lang="ts">
    let { status, message }: { status: number, message: string } = $props();
</script>

<div class="text-center p-8">
    <h1 class="text-3xl font-bold">Error {status}</h1>
    <p>{message}</p>
</div>
```

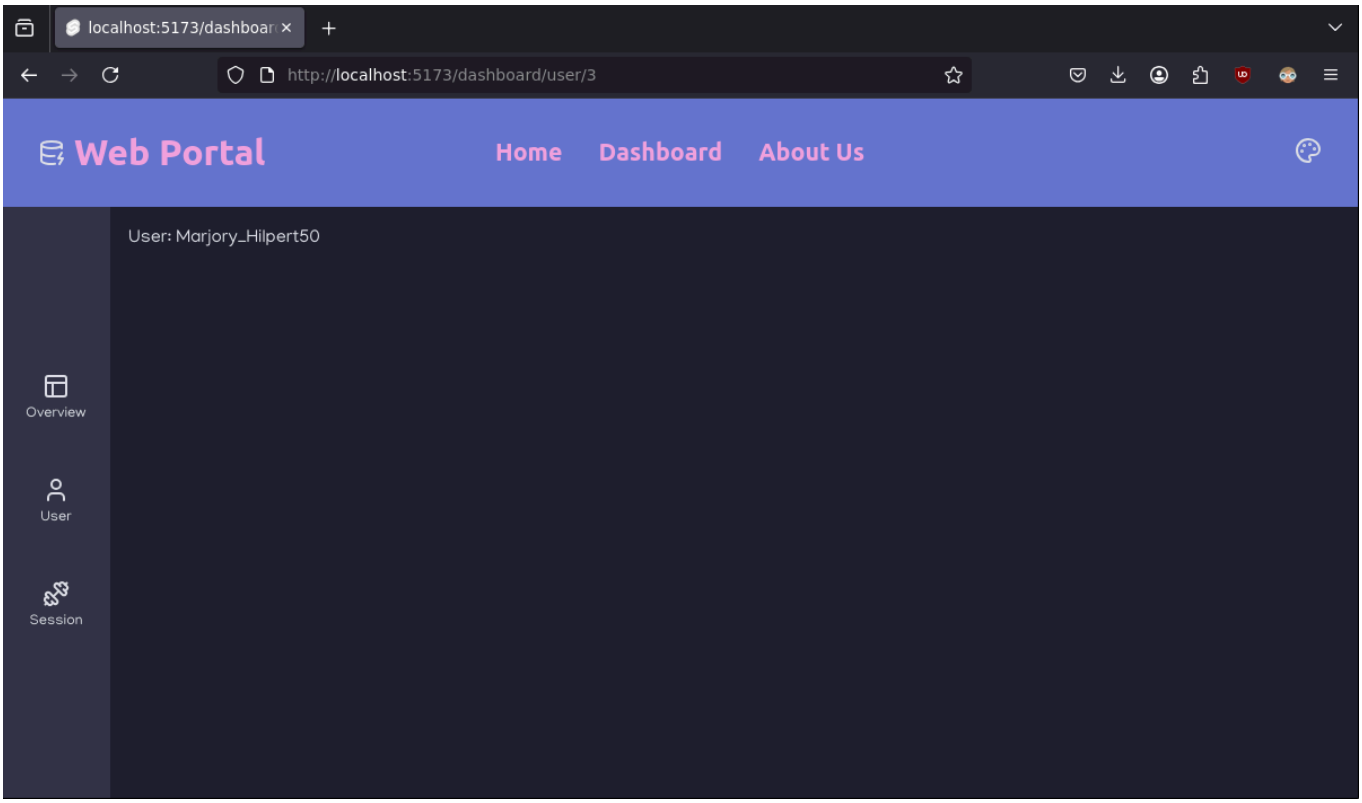
```
<!-- /src/routes/+error.svelte AND /src/routes/dashboard/+error.svelte -->
<script>
    import { page } from '$app/state';
    import Error from '$lib/ui/views/Error.svelte';
</script>

{#if page.error}
    <Error status={page.status} message={page.error.message}/>
{/if}
```

Let's add similar error handling to the `/dashboard/session/[id]` route, so that if a session with the given ID does not exist, we show a 404 error page.

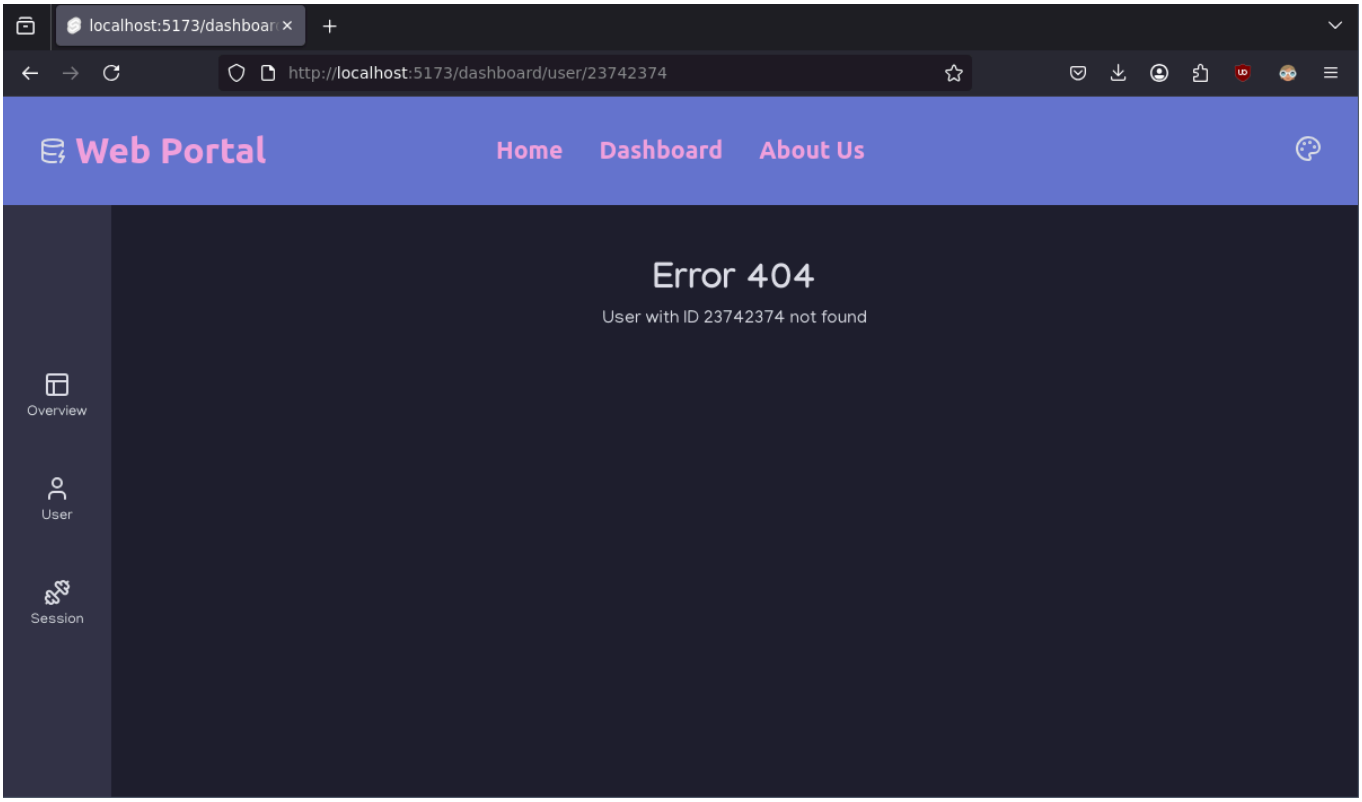
```
// /src/routes/dashboard/session/[id]/+page.server.ts

import type { PageServerLoad } from "./$types";

import type { Session} from "$lib/server/db/schema";
import { SessionDAO } from "$lib/server/dao/SessionDAO";
import { error } from "@sveltejs/kit";

export const load: PageServerLoad = async ({ params }) => {
    const id: number = parseInt(params.id);

    const session: Session | undefined = await
SessionDAO.getSessionById(id);

    if (!session) {
        console.error(`Session with ID ${id} not found. Showing 404.`);
        throw error(404, `Session with ID ${id} not found`);
    }

    return {
        session
    }
}
```

Okay, we're all set up to handle errors now. Let's look at how we're doing so far.

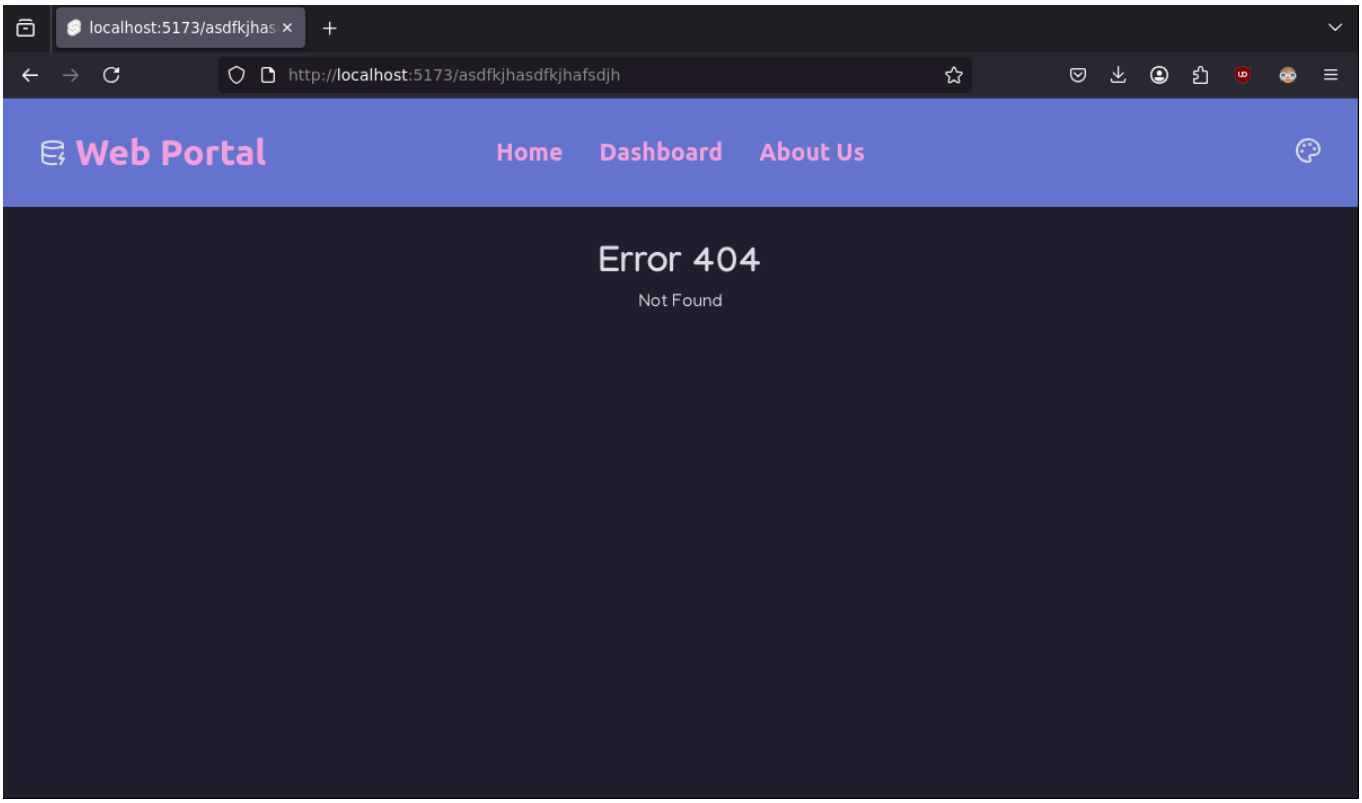
Querying an existing user shows their username:



And querying a non-existing user will lead us to an error page:



And - though perhaps a little less relevant for what we're doing - visiting any other non-existing route will also lead us to an error page, but this time not sharing the layout of the dashboard:



Visualizing Data

Okay, so we've loaded data into our pages, and handle errors when we encounter them. However, the way we display data is quite rudimentary right now. We want to create some components that visualize our data in a more sensible way.

Tables

We should be able to display most of our data in nice looking tables. So let's start with those.

Skeleton.dev already contains some neat looking Table components. We'll base our custom components on those, and add some extra functionality to them.

Let's start with looking at a simple `Table` example from Skeleton.Dev's documentation:

```
<script>
    const tableData = [
      { position: '0', name: 'Iron', symbol: 'Fe', atomic_no: '26' },
      { position: '1', name: 'Rhodium', symbol: 'Rh', atomic_no: '45' },
      { position: '2', name: 'Iodine', symbol: 'I', atomic_no: '53' },
      { position: '3', name: 'Radon', symbol: 'Rn', atomic_no: '86' },
      { position: '4', name: 'Technetium', symbol: 'Tc', atomic_no: '43' }
    ];
</script>
```

```
<div class="table-wrap">
  <table class="table caption-bottom">
    <tbody class="[&>tr]:hover:preset-tonal-primary">
      {
        tableData.map((row) => (
          <tr>
            <td>{row.position}</td>
            <td>{row.symbol}</td>
            <td>{row.name}</td>
            <td class="text-right">{row.atomic_no}</td>
          </tr>
        ))
      }
    </tbody>
  </table>
</div>
```

As with any Table, we have a `table` element that contains a `tbody` element with a list of `tr` (table row) elements, each containing a list of `td` (table data) elements. We can optionally add a `thead` or `tfoot` for a table header or footer respectively.

The above example uses the `map` function to iterate over the data in the `tableData` array, rendering the relevant data.

This looks pretty decent - but we can do better, to make the Table a little more flexibile. In the above example, the `td` fields all contain hard-coded references to the relevant data fields. We're instead going to make it so we can pass any type of data to our Table component, so we don't have to mess around with altering the Table code itself.

To do this, we'll export two new `interface`s in our `Table` component:

```
export interface TableData {
    caption?: string;
    columns: string[];
    rows: TableRow[];
}

export interface TableRow {
    values: (string | number | null)[];
    url?: string;
}
```

The `TableData` interface represents the data that we want to display in the Table, and contains an optional `caption` for the Table, a list of `columns` (the column headers), and a list of `rows`

(the actual data rows).
The `TableRow` interface represents a single row in the Table, and contains a list of `values` (the actual data values for that row), and an optional `url` to link to a specific page when the row is clicked.

Next, let's create a new `Table.svelte` component that uses these interfaces to render the Table:

```
<!-- /src/lib/ui/views/Table.svelte -->

<script lang="ts">
    import { goto } from '$app/navigation';

    export interface TableData {
        caption?: string;
        columns: string[];
        rows: TableRow[];
    }

    export interface TableRow {
        values: (string | number | null)[];
        url?: string;
    }

    let { table }: { table: TableData } = $props();
</script>

<section class="space-y-4">
    <!-- Table -->
    <div class="table-wrap">
        <table class="table caption-bottom">
            {#if table.caption}
                <caption style="white-space: pre-line">{table.caption}
</caption>
            {/if}
            <thead>
                <tr>
                    {#each table.columns as header, index (index)}
                        <th>{header}</th>
                    {/each}
                </tr>
            </thead>
            <tbody class="[&>tr]:hover:preset-tonal-primary">
                {#each table.rows as row, index (index)}
                    <tr
```

```
                        onclick={() => (row.url ? goto(row.url) : null)}
                        class={row.url ? 'cursor-pointer' : ''}
                    >
                        {#each row.values as value, i (i)}
                            <td>{value}</td>
                        {/each}
                    </tr>
                {/each}
            </tbody>
        </table>
    </div>
</section>
```

Key points to note here:

- We use the `TableData` and `TableRow` interfaces to define the data structure for the Table.
- We expose the `table` prop, which is of type `TableData`, so we can pass the Table data to the component.
- We use an `each` block to iterate over both the `columns` and `rows` arrays to render the Table headers and data rows.
- We use the `goto` function from `$app/navigation` to navigate to a specific page when a row is clicked, if the `url` property is set on the row.
- We add a `cursor-pointer` class to the row if it has a `url`, so that the user knows they can click on the row to navigate to a different page.

Next, let's see how we can use this `Table` component in our dashboard pages.

Showing user and session search results in tables

First, let's see how we can use it to display the found users in the `/dashboard/user/+page.svelte` file.

```
<!-- /src/routes/dashboard/user/+page.svelte -->

<script lang="ts">
    ...
    import Table from '$lib/ui/views/Table.svelte';

    ...

    let table = $derived({
        columns: ['Username', 'Created At'],
        rows: userResults?.map((user) => ({
            values: [user.username, user.createdAt.toLocaleDateString()],
```

```
            url: `/dashboard/user/${user.id}`
        }))
    });
</script>

<div class="...">
    ...
    {#if userResults && userResults.length > 0}
        <Card baseExtension="...">
            ...
            {#snippet article()}
                <div class="flex max-h-64 flex-col overflow-y-scroll">
                    <Table {table} />
                </div>
            {/snippet}
        </Card>
    {:else}
        ...
    {/if}
</div>
```

Key points to note here:

- We import the `Table` component and use it to render the Table with the found users.
- We create a `table` object that:
  - Contains the `columns` for the Table, which is an array of strings containing the column headers.
  - Contains the `rows` for the table, which is an array of `TableRow` objects.
    - Each `TableRow` object contains a `values` array with values for each column in the row, and an optional `url` to navigate to when the row is clicked.
  - We can do any additional data formatting / transformation here, such as string interpolation, date formatting, etc.

This will display the found users in a much more presentable way:

Similarly, we can implement the Table into our `/dashboard/session/+page.svelte` file to display the found sessions:
(We're not showing the full page code here, as you should be able to implement this yourself by now - the code is very similar to the user page.)

```
let table = $derived({
    columns: ['ID', 'Username'],
    rows: sessionResults?.map((session) => {
    return {
        values: [session.id, session.user.username],
        url: `/dashboard/session/${session.id}`
    };
    })
});
```

Displaying user sessions with an average score table

On our `/src/routes/dashboard/user/[id]` route, we'd not only like to show some data about the user itself (their `userProfile`) we'd like to show all of the sessions that user has played,

with a field showing their average score per session.

First, we need to write the relevant query in our `SessionDAO` and load the data in the `+page.server.ts` file:

```
// /src/lib/server/dao/SessionDao.ts
...
import { type Session, type User, scores, sessions, users } from
'$lib/server/db/schema';
import { avg, eq, like } from 'drizzle-orm';

export class SessionDAO extends DAO {
    ...

    static async findSessionsByUserId(userId: number):
Promise<SessionWithAverageScore[]> {
        const result = await DAO.db
            .select({
                session: sessions,
                averageScore: avg(scores.score).mapWith(Number)
            })
            .from(sessions)
            .innerJoin(scores, eq(sessions.id, scores.sessionId))
            .where(eq(sessions.userId, userId))
            .groupBy(sessions.id);

        return result.map((row) => ({
            ...row.session,
            averageScore: row.averageScore
        }));
    }
}

...

export type SessionWithAverageScore = Session & {
    averageScore: number;
};
```

Key points to note here:

- We use the `avg` function from Drizzle ORM to calculate the average score for each session.
- We use the `mapWith(Number)` method to ensure that the average score is returned as a number, as it is returned as a string by default.

- We use the `groupBy` method to group the results by session ID, so that we get one row per session with the average score.
- We export a new type `SessionWithAverageScore` that extends the `Session` type and adds an `averageScore` field.

While we're at it, let's also add a method to the `UserDAO` to find a users profile information by their ID, so we can load the user's profile information on the user detail page:

```
...
import type { User, UserProfile } from '$lib/server/db/schema';
import { userProfiles, users } from '$lib/server/db/schema';

export class UserDAO extends DAO {
    ...

    static async findUserWithProfileById(id: number):
Promise<UserWithProfile | undefined> {
        const result = await DAO.db
            .select({
                users,
                userProfiles
            })
            .from(users)
            .innerJoin(userProfiles, eq(users.id, userProfiles.userId))
            .where(eq(users.id, id));

        if (result.length !== 1) {
            return undefined;
        }

        return {
            ...result[0].users,
            profile: result[0].userProfiles
        };
    }
}

export type UserWithProfile = User & {
    profile: UserProfile;
};
```

Key points to note here:

- We use the `innerJoin` method to join the `users` and `userProfiles` tables, so we can

get the user's profile information.

- We perform a sanity check to ensure that we only return a single user with their profile information (our database constraints should enforce this, but it's good practice to check).
- We export a new type `UserWithProfile` that extends the `User` type and adds a `profile` field containing the user's profile information.

Next, we can load this data in our `+page.server.ts` file for the user detail page:

```
// /src/routes/dashboard/user/[id]/+page.server.ts

import type { PageServerLoad } from './$types';

import { UserDAO, type UserWithProfile } from '$lib/server/dao/UserDAO';
import { error } from '@sveltejs/kit';
import { SessionDAO, type SessionWithAverageScore } from
'$lib/server/dao/SessionDAO';

export const load: PageServerLoad = async ({ params }) => {
    const id: number = parseInt(params.id);

    const user: UserWithProfile | undefined = await
UserDAO.findUserWithProfileById(id);
    const sessionsByUser: SessionWithAverageScore[] = await
SessionDAO.findSessionsByUserId(id);


    if (!user) {
        console.error(`User with ID ${id} not found. Showing 404.`);
        throw error(404, `User with ID ${id} not found`);
    }

    return {
        user,
        sessionsByUser
    };
};
```

And finally, we can display this data in our `+page.svelte` file for the user detail page.

```
<!-- /src/routes/dashboard/user/[id]/+page.svelte -->

<script lang="ts">
    import type { UserWithProfile } from '$lib/server/dao/UserDAO';
    import type { SessionWithAverageScore } from
```

```
'$lib/server/dao/SessionDAO';
    import type { TableData } from '$lib/ui/views/Table.svelte';
    import Card from '$lib/ui/views/Card.svelte';
    import Table from '$lib/ui/views/Table.svelte';

    let { data } = $props();

    let user: UserWithProfile = $derived(data.user);

    const table: TableData = $derived({
        caption: 'Scores in this session.',
        columns: ['ID', 'Duration', 'Created At', 'Ended At', 'Average
Score'],
        rows: data.sessionsByUser.map((session: SessionWithAverageScore) =>
({
            values: [
                session.id,
                session.duration,
                session.createdAt.toLocaleString(),
                getSessionEnd(session.createdAt,
session.duration).toLocaleString(),
                session.averageScore
            ],
            url: `/dashboard/session/${session.id}`
        }))
    });

    function getSessionEnd(createdAt: Date, duration: number): Date {
        return new Date(createdAt.getTime() + duration * 1000);
    }

    function getCurrentAge(dateOfBirth: Date): number {
        const now = new Date();
        let age = now.getFullYear() - dateOfBirth.getFullYear();
        const monthDiff = now.getMonth() - dateOfBirth.getMonth();
        if (monthDiff < 0 || (monthDiff === 0 && now.getDate() <
dateOfBirth.getDate())) {
            age--;
        }
        return age;
    }

    const userAge = $derived(getCurrentAge(user.dateOfBirth));

</script>
```

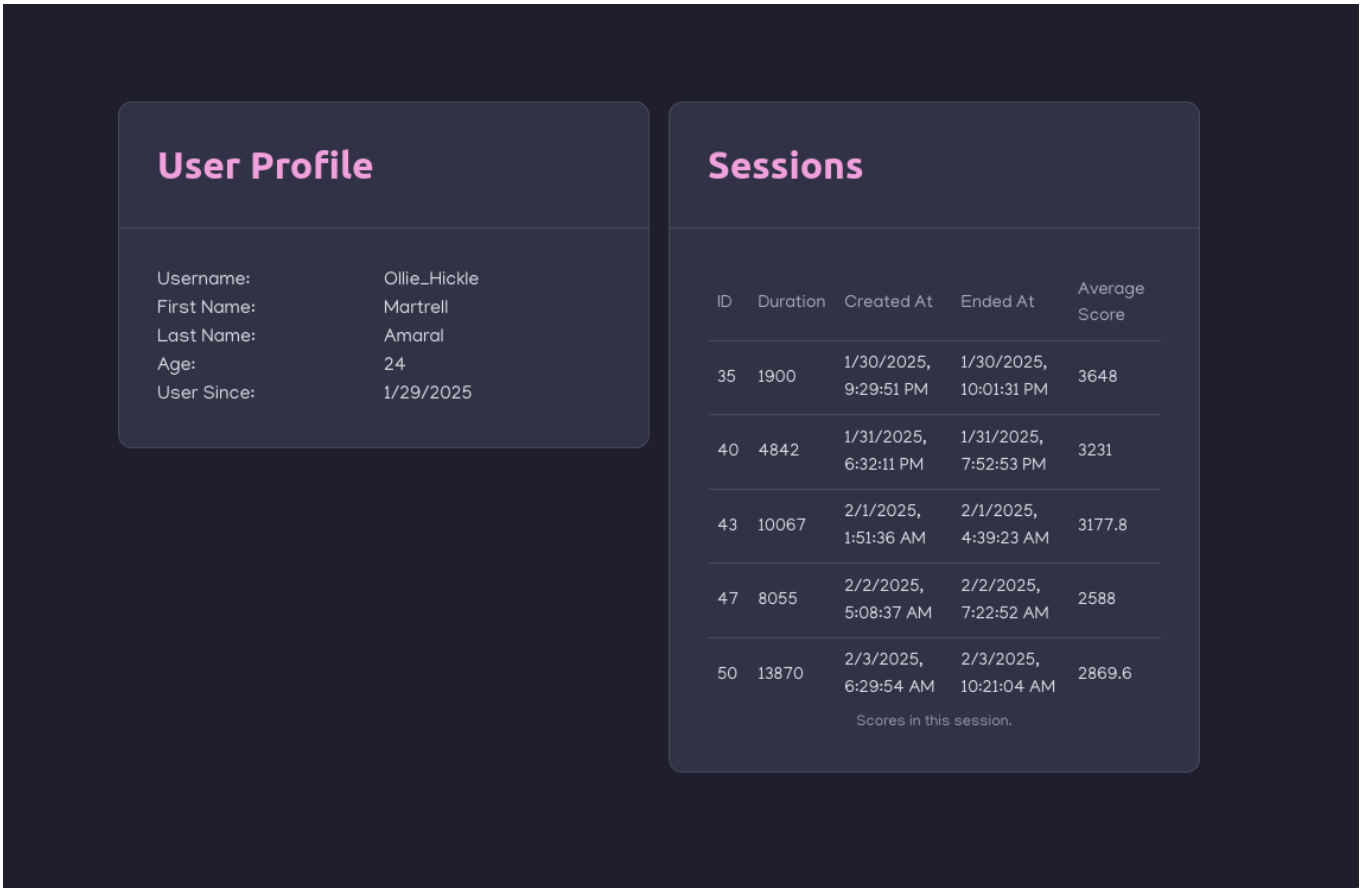
```
<div class="m-auto grid grid-cols-1 gap-4 lg:grid-cols-2">
    <Card>
        {#snippet header()}
            <h1>User Profile</h1>
        {/snippet}
        {#snippet article()}
            <div class="grid grid-cols-2 justify-between">
                <span>Username:</span>
                <span>{user.username}</span>
                <span>First Name:</span>
                <span>{user.profile.firstName}</span>
                <span>Last Name:</span>
                <span>{user.profile.lastName}</span>
                <span>Age:</span>
                <span>{userAge}</span>
                <span>User Since:</span>
                <span>{user.createdAt.toLocaleDateString()}</span>
            </div>
        {/snippet}
    </Card>

    <Card>
        {#snippet header()}
            <h1>Sessions</h1>
        {/snippet}
        {#snippet article()}
            <Table {table} />
        {/snippet}
    </Card>
</div>
```

Key points to note here:

- We import the `UserWithProfile` and `SessionWithAverageScore` types to type the data we receive from the `load` function.
- We create a `TableData` object that contains the data for the Table, including the caption, columns, and rows.
    - We use the `getSessionEnd` function to calculate the end time of the session based on the start time and duration.
    - We make the rows clickable by providing a `url` for each row, which navigates to the session detail page when clicked.
- We add a User Profile card that displays the user's profile information
    - We use the `getCurrentAge` function to calculate the user's age based on their date of birth.

- We display the user's sessions in a Table, using the `Table` component we created earlier

We should now see a neat User Profile section and a table providing an overview of their sessions, along with the average score per session:



Displaying scores per session in a table

Let's display some information about a session, and the scores achieved in that session. Our workflow will be similar to the previous section, but instead of displaying information about a users' sessions, we will display information about a single session, and the scores achieved in that session.

First, we'll expand our `SessionDAO` to include a method that retrieves a session, along with some information about the user who created it (previously we only retrieved the session itself).

```
// /src/lib/server/dao/SessionDao.ts
export class SessionDAO extends DAO {
    ...

    static async getSessionByIdWithUser(id: number): Promise<SessionWithUser
| undefined> {
        const result = await DAO.db.select({
            session: sessions,
```

```
            user: users
        })
            .from(sessions)
            .innerJoin(users, eq(sessions.userId, users.id))
            .where(eq(sessions.id, id))
            .limit(1)
            .then((rows) => rows[0]);
        return result ? { ...result.session, user: result.user } :
undefined;
    }
}
```

Next, we'll expand the ScoreDAO to include a method that retrieves all scores for a given session:

```
// /src/lib/server/dao/ScoreDAO.ts
export class ScoreDAO extends DAO {
    ...

    static async findScoresForSession(sessionId: number): Promise<Score[]> {
        return await DAO.db.query.scores.findMany({
            where: eq(scores.sessionId, sessionId)
        });
    }
}
```

We'll then load this data in our `+page.server.ts` file for the session detail page:

```
// /src/routes/dashboard/session/[id]/+page.server.ts

...

import type { Score } from "$lib/server/db/schema";
import { SessionDAO, type SessionWithUser } from
"$lib/server/dao/SessionDAO";
import { ScoreDAO } from "$lib/server/dao/ScoreDAO";

export const load: PageServerLoad = async ({ params }) => {
    ...

    const session: SessionWithUser | undefined = await
SessionDAO.getSessionByIdWithUser(id);
    const scores: Score[] = await ScoreDAO.findScoresForSession(id);

    ...
```

```
    return {
        session,
        scoresInSession: scores,
    }
}
```

And finally, we'll display this data in our `+page.svelte` file for the session detail page:

```
<!-- /src/routes/dashboard/session/[id]/+page.svelte -->

<script lang="ts">
    import type { TableData } from '$lib/ui/views/Table.svelte';
    import Card from '$lib/ui/views/Card.svelte';
    import Table from '$lib/ui/views/Table.svelte';
    import type { Score } from '$lib/server/db/schema';
    import type { SessionWithUser } from '$lib/server/dao/SessionDAO.js';

    let { data } = $props();

    const session: SessionWithUser = $derived(data.session);
    const user = $derived(session.user);

    const table: TableData = $derived({
        caption: 'A list of scores in this session.',
        columns: ['Level ID', 'Score', 'Accuracy', 'Time Taken', 'Created
At'],
        rows: data.scoresInSession.map((score: Score) => ({
            values: [
                score.levelId,
                score.score,
                score.accuracy,
                score.timeTaken,
                score.createdAt.toLocaleString()
            ]
        }))
    });

    function getSessionEnd(createdAt: Date, duration: number): Date {
        return new Date(createdAt.getTime() + duration * 1000);
    }

    const sessionEnd = $derived(getSessionEnd(session.createdAt,
session.duration));
</script>
```

```
<div class="m-auto grid grid-cols-1 gap-4 lg:grid-cols-2">
    <Card>
        {#snippet header()}
            <h1>Session Information</h1>
        {/snippet}
        {#snippet article()}
            <div class="grid grid-cols-2 justify-between">
                <span>ID</span>
                <span>{session.id}</span>
                <span>Created At</span>
                <span>{session.createdAt.toLocaleString()}</span>
                <span>Ended At</span>
                <span>{sessionEnd.toLocaleString()}</span>
                <span>By</span>
                <a href="/dashboard/user/{user.id}" class="underline">
{user.username}</a>
            </div>
        {/snippet}
    </Card>

    <Card>
        {#snippet header()}
            <h1>Scores</h1>
        {/snippet}
        {#snippet article()}
            <Table {table} />
        {/snippet}
    </Card>
</div>
```

Key differences from the previous section:

- We display the session information in a Card, instead of a User Profile.
- We display the scores in a Table, instead of the user's sessions.
- We no longer have any clickable rows in the Table
- We add a clickable link to the user who created the session, which navigates to the user's detail page.

Data Util functions

We introduced some duplicate code in the previous sections to visualize our data, such as the `getSessionEnd` and `getCurrentAge` functions.
We can extract these into a separate utility file to avoid duplication and make our code cleaner.

We'll create a new file `/src/lib/utils/date.ts` and move the `getSessionEnd` and `getCurrentAge` functions there, giving them a more generic name to reflect their utility nature:

```
// /src/lib/utils/date.ts

export function dateAddSeconds(createdAt: Date, duration: number): Date {
    return new Date(createdAt.getTime() + duration * 1000);
}

export function ageFromDateOfBirth(dateOfBirth: Date): number {
    const now = new Date();
    let age = now.getFullYear() - dateOfBirth.getFullYear();
    const monthDiff = now.getMonth() - dateOfBirth.getMonth();
    if (monthDiff < 0 || (monthDiff === 0 && now.getDate() <
dateOfBirth.getDate())) {
        age--;
    }
    return age;
}
```

And then we use them in the `+page.svelte` files for the user and session detail pages:

```
<!-- /src/routes/dashboard/user/[id]/+page.svelte -->

<script lang="ts">
    ...

    import { ageFromDateOfBirth, dateAddSeconds } from '$lib/utils/date';

    ...
    const table: TableData = $derived({
        caption: ...,
        columns: ...,
        rows: data.sessionsByUser.map((session: SessionWithAverageScore) =>
({
            values: [
                ...,
                dateAddSeconds(session.createdAt,
session.duration).toLocaleString(),
                ...
            ],
            ...
        }))
    });
```

```
    const userAge = $derived(ageFromDateOfBirth(user.dateOfBirth));
</script>
```

```
<!-- /src/routes/dashboard/session/[id]/+page.svelte -->

<script lang="ts">
    ...

    import { dateAddSeconds } from '$lib/utils/date';

    ...

    const sessionEnd = $derived(dateAddSeconds(session.createdAt,
session.duration));
</script>

...
```

Now our page code doesn't need to worry about the implementation details of how the session end time or user age is calculated, and we can reuse these utility functions in other parts of our application as well.

Table Pagination

As sessions and users grow in number over time, we may want to implement pagination for our tables, so that a user isn't overwhelmed by too much data at once.
We can implement this by adding the Skeleton `pagination` component.

First we'll implement some typescript for the pagination functionality:

- import the component and some icons from `lucide-svelte`
- define a `PaginationOptions` interface to configure the pagination behavior
- add the `paginationOptions` property to the `TableData` interface
- define some default values for the pagination options
- define some reactive state variables to hold the current page, page size, and whether pagination is enabled
- created a derived sliced array of the source data, which holds the content that should be displayed on the current page

```
<!-- /src/lib/ui/views/Table.svelte -->
<script lang="ts">
```

```
    ...

    import { Pagination } from '@skeletonlabs/skeleton-svelte';
    import { ArrowLeft, ArrowRight, ChevronFirst, ChevronLast, Ellipsis }
from 'lucide-svelte';

    export interface TableData {
        ...,
        paginationOptions?: PaginationOptions;
    }

    ...

    export interface PaginationOptions {
        enabled?: boolean;
        page?: number;
        size?: number;
        sizePerPage?: number[];
    }

    const paginationDefaults: PaginationOptions = {
        enabled: true,
        page: 1,
        size: 10,
        sizePerPage: [5, 10]
    };

    let { table }: { table: TableData } = $props();


    // State
    // We pick the pagination options from the table, or use the defaults if
not provided
    let page = $state(table.paginationOptions?.page ??
paginationDefaults.page!);
    let size = $state(table.paginationOptions?.size ??
paginationDefaults.size!);
    let enabled = $state(table.paginationOptions?.enabled ??
paginationDefaults.enabled!);
    let sizePerPage = $state(table.paginationOptions?.sizePerPage ??
paginationDefaults.sizePerPage!);

    const slicedSource = $derived((source: TableRow[]) =>
        enabled ? source.slice((page - 1) * size, page * size) : source
    );
</script>
```

Next, we'll alter the `Table` markup to:

- loop over the `slicedSource` rather than the full `table.rows` to only display the rows for the current page
- expand the `Table` markup to include the pagination controls, which will be displayed in the footer of the table
    - in the `onchange` events we update the state variables, which will trigger a re-render of the component with the new page or page size

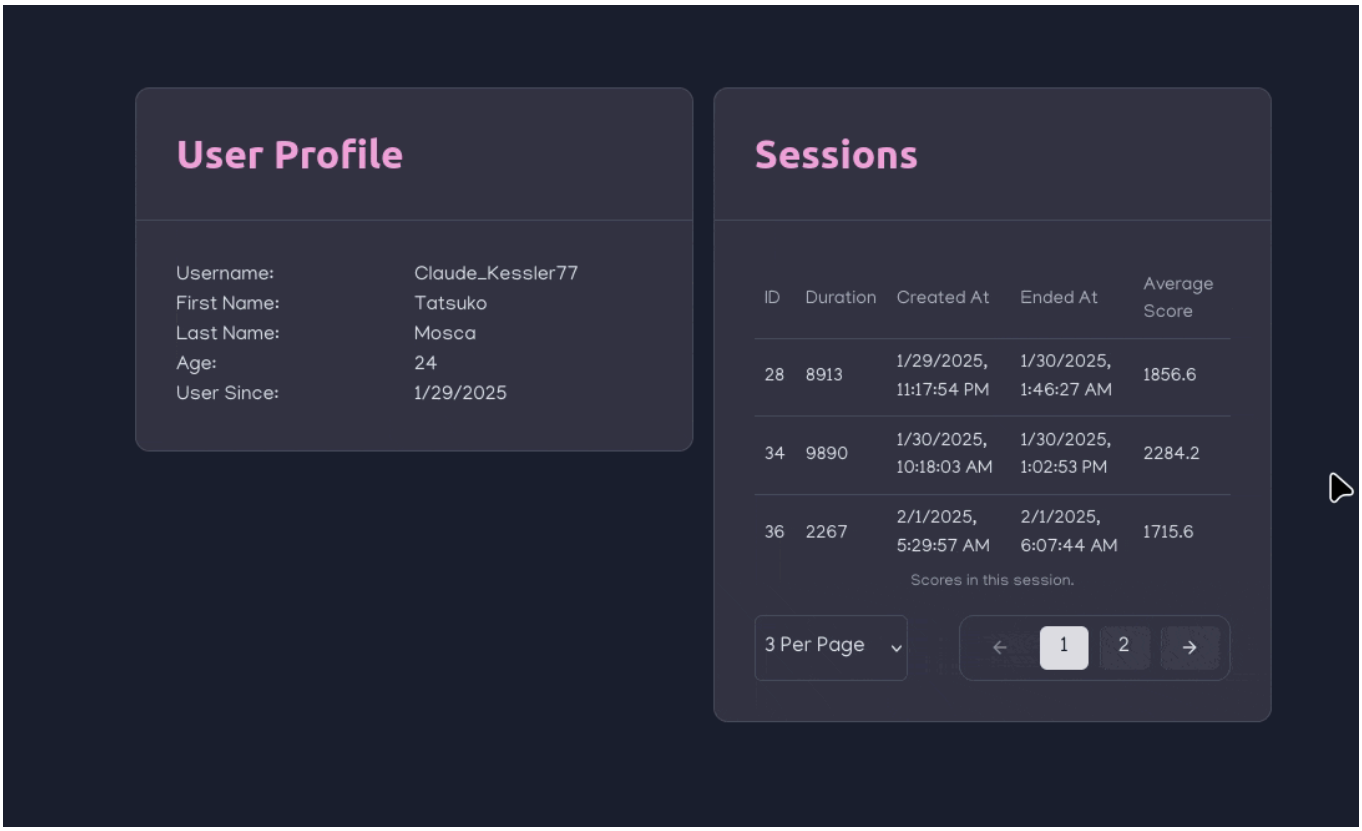
```
<!-- /src/lib/ui/views/Table.svelte -->

<section class="...">
    <!-- Table -->
    <div class="...">
        <table class="...">
            ...
            <tbody class="...">
                {#each slicedSource(table.rows) as row, index (index)}
                    <tr
                        onclick={() => (row.url ? goto(row.url) : null)}
                        class={row.url ? 'cursor-pointer' : ''}
                    >
                        {#each row.values as value, i (i)}
                            <td>{value}</td>
                        {/each}
                    </tr>
                {/each}
            </tbody>
        </table>
    </div>
    {#if enabled}
        <!-- Footer -->
        <footer class="flex justify-between">
            <select
                name="size"
                id="size"
                class="select w-fit max-w-[150px] px-2"
                value={size}
                onchange={(e) => (size = Number(e.currentTarget.value))}
            >
                {#each sizePerPage as v, i (i)}
                    <option value={v}>{v} Per Page</option>
                {/each}
                <option value={table.rows.length}>Show All</option>
            </select>
```

```
            <!-- Pagination -->
            <Pagination
                data={table.rows}
                {page}
                onPageChange={(e) => (page = e.page)}
                pageSize={size}
                onPageSizeChange={(e) => (size = e.pageSize)}
                siblingCount={4}
            >
                {#snippet labelEllipsis()}<Ellipsis class="size-4" />
{/snippet}
                {#snippet labelNext()}<ArrowRight class="size-4" />
{/snippet}
                {#snippet labelPrevious()}<ArrowLeft class="size-4" />
{/snippet}
                {#snippet labelFirst()}<ChevronFirst class="size-4" />
{/snippet}
                {#snippet labelLast()}<ChevronLast class="size-4" />
{/snippet}
            </Pagination>
        </footer>
    {/if}
</section>
```

Now, to use the pagination functionality, we can simply pass the `paginationOptions` property to the `TableData` object.

For example, to set our Session table in `/src/routes/dashboard/user/[id]/+page.svelte` to only show 3 sessions per page, we can do the following:

```
<script lang="ts">
    ...

    const table: TableData = $derived({
        ...,
        paginationOptions: {
            // Show 3 sessions per page by default
            size: 3,
            // Pass 3 as an option for the dropdown that allows the user to
change the page size
            sizePerPage: [3, 5, 10],
        }
    });
</script>
```



We can also choose to disable the feature we just added for specific tables by setting the `enabled` property to `false` in the `paginationOptions`.
For example, to disable pagination on the user and session search results:

```
<!-- /src/routes/dashboard/user/+page.svelte AND
/src/routes/dashboard/session/+page.svelte -->
<script lang="ts">
    ...

    let table = $derived({
        ...
        paginationOptions: {
            enabled: false
        }
    });
</script>
```

Charts

Next, let's look at visualizing our data in a more graphical way, using charts.

To do this, we'll use the Chart.js library, which is a popular JavaScript library for creating charts and graphs.

We won't go into too much detail about how to use Chart.js, as it's website is well documented, but we'll mainly talk about how we can integrate it into our SvelteKit application.

We want to create a Chart in which Time is on the X-axis, to visualize the scores achieved in a session over time. For this we need a date adapter.
We'll use a date-fns adapter to handle date formatting in the charts.

First, we'll install `chart.js`, `date-fns`, and the `chartjs-adapter-date-fns` package:

```
npm i chart.js date-fns chartjs-adapter-date-fns
```

Next, let's create a new `Chart.svelte` component in `/src/lib/ui/views/charts/Chart.svelte`, that we will use as a base component for our charts:

```
<!-- /src/lib/ui/views/charts/Chart.svelte -->

<script lang="ts">
    //@ts-nocheck
    import { Chart } from 'chart.js/auto';
    import 'chartjs-adapter-date-fns';
    import { onMount } from 'svelte';

    let { type, data, options } = $props();

    let chartElement;
    let chartInstance: Chart | null = null;

    onMount(() => {
        renderChart();
        window.addEventListener('resize', renderChart);
    });

    function renderChart() {
        // When resizing, it's possible the chart gets stuck in it's
smallest state.
        // To avoid this, we destroy the chart instance and create a new
one, disabling the pop-in animation.
        let animate = true;
        if (chartInstance) {
            chartInstance.destroy();
            animate = false;
        }
        chartInstance = new Chart(chartElement, {
```

```
            type: type,
            data: data,
            options: {
                ...options,
                animation: animate
            }
        });
    }
</script>

<canvas id="chart" bind:this={chartElement}></canvas>
```

The key points to note here are:

- We import the `Chart` class from `chart.js/auto`
- We expose the `type`, `data`, and `options` props, which will be used to configure the chart.
- We use the `onMount` lifecycle function to render the chart when the component is mounted.
- We bind the `<canvas>` element to `chartElement`, which is where the chart will be rendered.
- We import `chartjs-adapter-date-fns` to enable date formatting in the chart.
- We add some extra logic to handle resizing the chart to ensure it resizes correctly when the window is resized.

Next, let's create a new `ScoreOverTimeInSessionChart.svelte` component in `/src/lib/ui/views/charts/ScoreOverTimeInSessionChart.svelte`, which will use the `Chart` component to display the scores achieved in a session over time.
This won't be a reusable component, but we can abstract away all the logic regarding this specific chart, so we can easily use it in our session detail page.

The main thing we need to understand is what type of chart we want to create, and how to `chart.js` expects the data to be formatted.
In this case, we want to create a line chart, where the x-axis is a time axis, and the y-axis is the score achieved at that time.
We can find examples of line charts on the Chart.js documentation website, and learn from there.

Chart.js expects the data to be in a specific format, of x and y values.
We map the score data to a new array of objects, where each object contains an `x` value (the time the score was achieved) and a `y` value (the score itself).

```
<!-- /src/lib/ui/views/charts/ScoreOverTimeInSessionChart.svelte -->

<script lang="ts">
    import type { Score } from '$lib/server/db/schema';
    import { nl } from 'date-fns/locale';
    import Chart from '$lib/ui/views/charts/Chart.svelte';

    let { scores }: { scores: Score[] } = $props();

    // We sort the scores by createdAt,
    // then map the x values (time) to the createdAt,
    // and the y values (score) to the score.
    const inData = scores
        .sort((a, b) => a.createdAt.getTime() - b.createdAt.getTime())
        .map((score) => {
            return {
                x: score.createdAt,
                y: score.score
            };
        });

    const chartData = {
        // Display a line chart
        type: 'line',

        data: {
            // Labels for the x-axis
            labels: inData.map((score) => score.x),
            // Datasets for the y-axis
            datasets: [
                // Just one dataset: the score over time
                {
                    label: 'Score',
                    data: inData.map((score) => score.y)
                }
            ]
        },
        options: {
            // Define how the charts scales should behave
            scales: {
                // The x-axis is of type 'time' and has a date adapter for
the 'nl' locale
                x: {
                    type: 'time',
                    adapters: {
                        date: {
```

```
                            locale: nl
                        }
                    },
                    bounds: 'data'
                },
                // The y-axis starts at zero (rather than the lowest value)
                y: {
                    beginAtZero: true
                }
            }
        }
    };
</script>

<Chart type={chartData.type} data={chartData.data} options=
{chartData.options} />
```

Key points to note here:

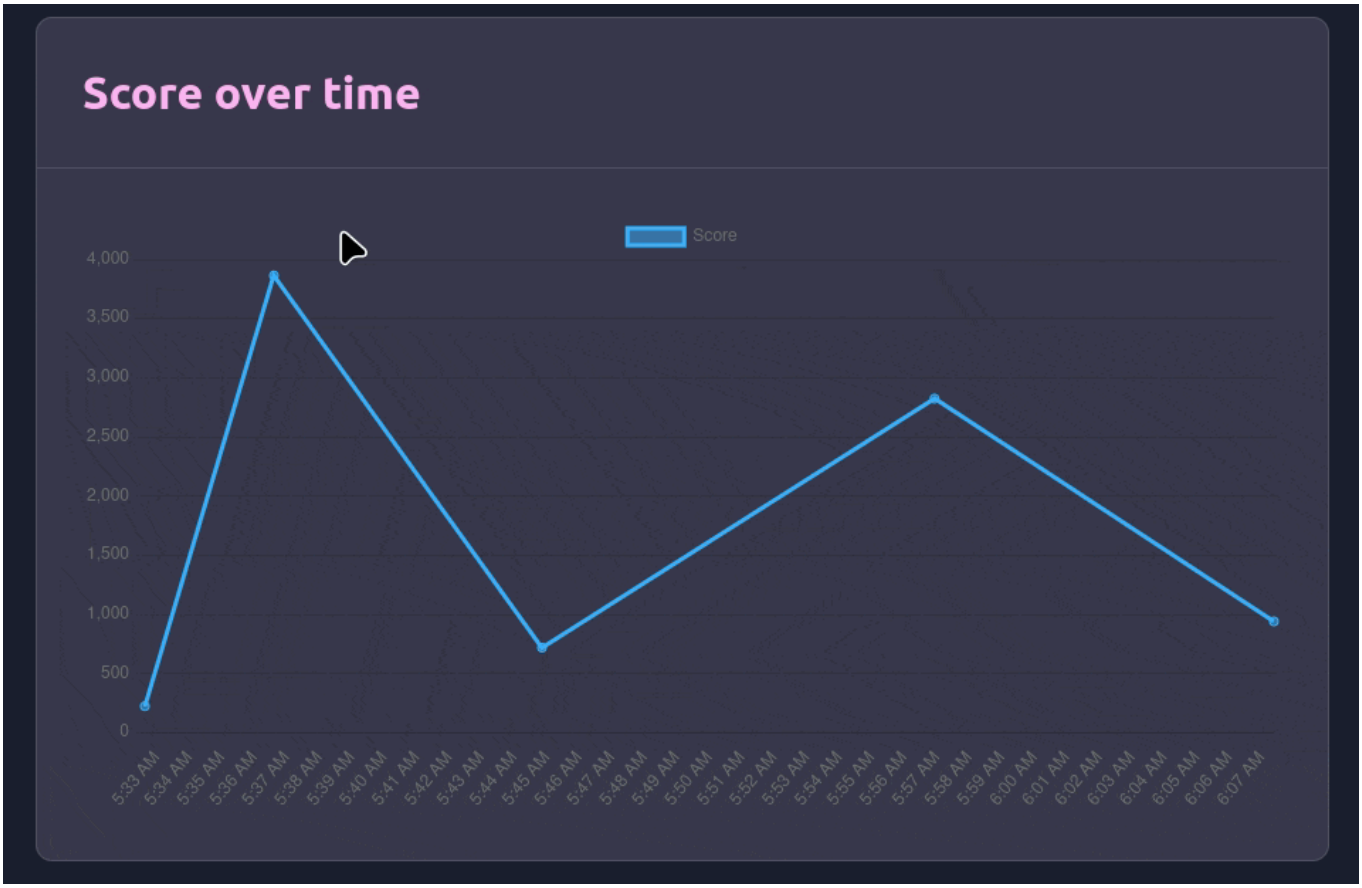
- We import the `Chart` component we created earlier, and use it to render the chart.
- We expose the `scores` prop, which is an array of `Score` objects.
- We sort the scores by `createdAt`, and map the `x` values to the `createdAt` date, and the `y` values to the `score`.
- We create a `chartData` object that contains the data for the chart, including the labels for the x-axis and the datasets for the y-axis.
- We define the chart options, including the x-axis type as `time`, and the date adapter for the `nl` locale (Dutch).

Next, we can use this `ScoreOverTimeInSessionChart` component in `/src/routes/dashboard/session/[id]/+page.svelte` to display the scores achieved in a session over time:

```
<!-- /src/routes/dashboard/session/[id]/+page.svelte -->

<script lang="ts">
    ...
    import ScoreOverTimeInSessionChart from
'$lib/ui/views/charts/ScoreOverTimeInSessionChart.svelte';

    ...
</script>

<div class="m-auto grid grid-cols-1 gap-4 lg:grid-cols-2">
    <Card>
```

```
        ...
    </Card>

    <Card>
        ...
    </Card>
    <Card baseExtension="lg:col-span-2 !max-w-full">
        {#snippet header()}
            <h1>Score over time</h1>
        {/snippet}
        {#snippet article()}
            <div class="justify-center">
                <ScoreOverTimeInSessionChart scores={data.scoresInSession}
/>
            </div>
        {/snippet}
    </Card>
</div>
```

And just like that, we have a neat chart mapping the score over time in a session!



To ensure this wasn't a fluke, let's also implement a chart for the user detail page, showing the average score per session over time.

We'll create a new `AverageScoreOverTimeForUserChart.svelte` component in `/src/lib/ui/views/charts/AverageScoreOverTimeForUserChart.svelte`.
This is largely a duplicate of the previous chart, but we'll be using the `SessionWithAverageScore` type instead of the `Score` type, and we'll be mapping the `averageScore` to the `y` value instead of the `score` (and altering it's label).

```
<script lang="ts">
    ...
    import type { SessionWithAverageScore } from
'$lib/server/dao/SessionDAO';

    let { scores: sessions }: { scores: SessionWithAverageScore[] } =
$props();

    // We sort the scores by createdAt,
    // then map the x values to the createdAt,
    // and the y values to the score.
    const inData = sessions
        .sort((a, b) => a.createdAt.getTime() - b.createdAt.getTime())
        .map((score) => {
            return {
                x: score.createdAt,
                y: score.averageScore
            };
        });

    const chartData = {
        ...,
        data: {
            ...,
            datasets: [
                {
                    label: 'Average Score',
                    data: inData.map((score) => score.y)
                }
            ]
        },
        ...
    };
</script>

<Chart type={chartData.type} data={chartData.data} options=
{chartData.options} />
```

Adding it to the user detail page is similar to the previous chart, so we won't go into too much detail here either:

```
<!-- /src/routes/dashboard/user/[id]/+page.svelte -->

<script lang="ts">
    ...
    import AverageScoreOverTimeForUserChart from
'$lib/ui/views/charts/AverageScoreOverTimeForUserChart.svelte';
    ...
</script>

<div class="...">
    <Card>
        ...
    </Card>

    <Card>
        ...
    </Card>
    <Card baseExtension="lg:col-span-2 !max-w-full">
        {#snippet header()}
            <h1>Average score over time</h1>
        {/snippet}
        {#snippet article()}
            <div class="justify-center">
                <AverageScoreOverTimeForUserChart scores=
{data.sessionsByUser} />
            </div>
        {/snippet}
    </Card>
</div>
```

Retouching the Dashboard Overview Page

Most of our pages look pretty good now, but we haven't retouched our dashboard overview page yet.

Let's do that now to wrap this chapter up.
Along with our `TopScorers` list we will add a list of all users in the database.
Furthermore, we will use our new Table component to display these lists in a more presentable way.

First, let's expand the `/src/routes/dashboard/page.server.ts` file to query the database for all users (we've already implemented the relevant query in the UserDAO):

```
// /src/routes/dashboard/+page.server.ts

...
import { UserDAO } from '$lib/server/dao/UserDAO';
import type { User } from '$lib/server/db/schema';

export const load: PageServerLoad = async () => {
    const limit: number = 10;
    const users: User[] = await UserDAO.getAllUsers();
    const topScorers: TopScorer[] = await ScoreDAO.getTopScorers(limit);

    return {
        users,
        topScorers
    };
};
```

And then update the `/src/routes/dashboard/+page.svelte` file to display the users in a Table, along with the top scorers:

```
<!-- /src/routes/dashboard/+page.svelte -->

<script lang="ts">
    import type { TopScorer } from '$lib/server/dao/ScoreDAO';
    import type { User } from '$lib/server/db/schema';
    import Card from '$lib/ui/views/Card.svelte';
    import type { TableData } from '$lib/ui/views/Table.svelte';
    import Table from '$lib/ui/views/Table.svelte';
    import { ageFromDateOfBirth } from '$lib/utils/date.js';

    const { data } = $props();

    const topScoreTable: TableData = $derived({
        caption:
            'A list of the top 10 users who have the highest scores.\nClick
to view the session in which they achieved it.',
        columns: [
            'Ranking',
            'Username',
            'Level ID',
            'Score',
            'Accuracy',
```

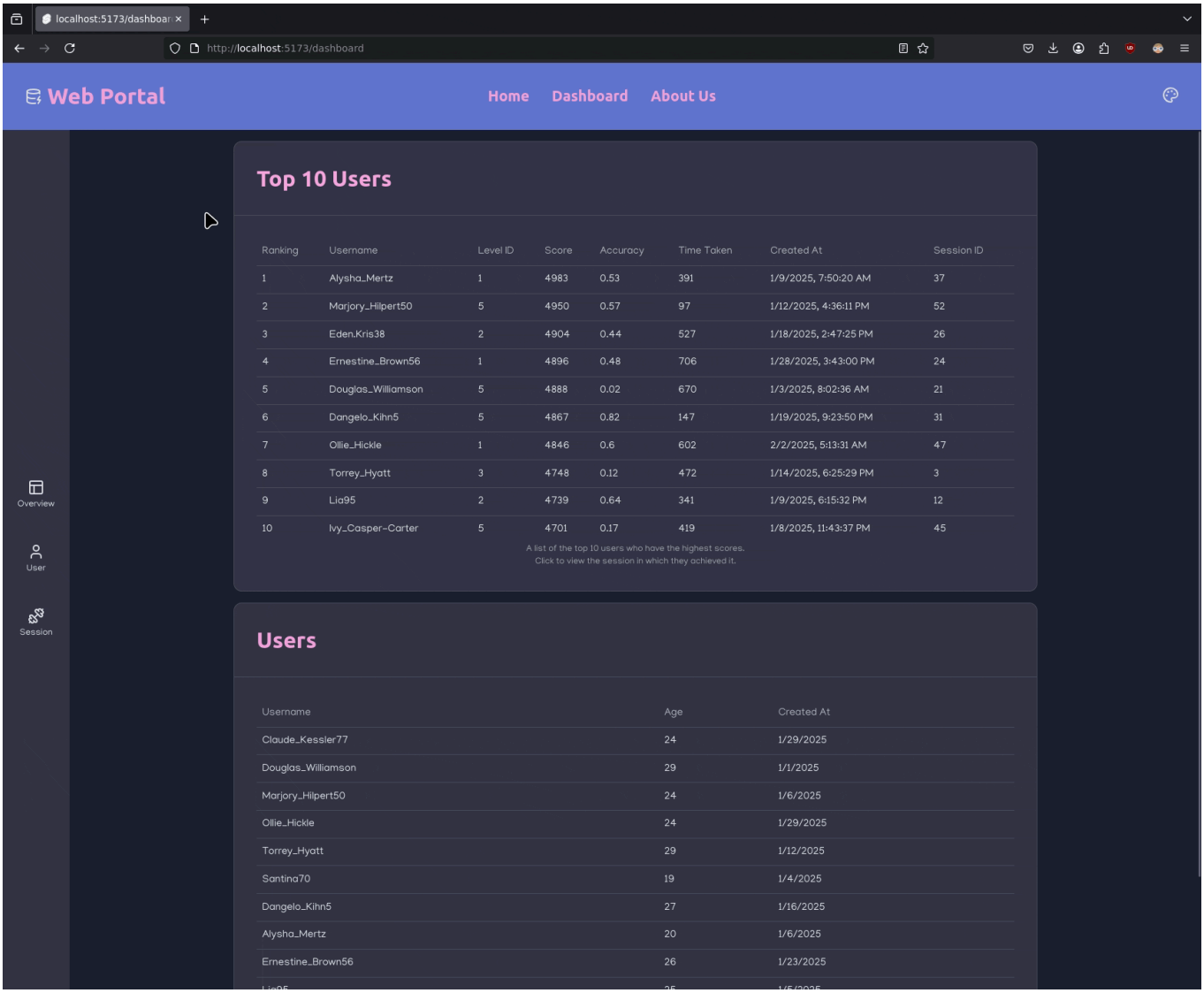
```
            'Time Taken',
            'Created At',
            'Session ID'
        ],
        rows: data.topScorers.map((score: TopScorer, index: number) => ({
            values: [
                index + 1,
                score.user.username,
                score.score.levelId,
                score.score.score,
                score.score.accuracy,
                score.score.timeTaken,
                score.score.createdAt.toLocaleString(),
                score.session.id
            ],
            url: `/dashboard/session/${score.session.id}`
        })),
        paginationOptions: { enabled: false }
    });

    const userTable: TableData = $derived({
        caption: 'A list of users.\nClick to view their profile.',
        columns: ['Username', 'Age', 'Created At'],
        rows: data.users.map((user: User) => ({
            values: [
                user.username,
                ageFromDateOfBirth(user.dateOfBirth),
                user.createdAt.toLocaleDateString()
            ],
            url: `/dashboard/user/${user.id}`
        }))
    });
</script>

<div class="mx-auto grid grid-cols-1 gap-y-4">
    <Card baseExtension="!max-w-full w-6xl">
        {#snippet header()}
            <h1>Top 10 Users</h1>
        {/snippet}
        {#snippet article()}
            <Table table={topScoreTable} />
        {/snippet}
    </Card>
    <Card baseExtension="!max-w-full w-6xl">
        {#snippet header()}
            <h1>Users</h1>
```

```
        {/snippet}
        {#snippet article()}
            <Table table={userTable} />
        {/snippet}
    </Card>
</div>
```

Key points to note here:

- We create two `TableData` objects, one for the top scorers and one for the users.
- We disable pagination for the top scorers table, as we only want to show the top 10 users.
- We add an index to the top scorers table to display the ranking of each user.
- We set the `url` property for each row, so that clicking on a row navigates to the relevant user (usertable) or session detail page in which the top score was achieved (top scorers table).
- We use the `ageFromDateOfBirth` utility function to calculate the age of the user based on their date of birth.

Cool. We now have a nice overview page that takes us to the user detail page when clicking on a user, and to the session detail page when clicking on a top scorer.



Retouching the About Us section

This section isn't really about data visualization, but rather about polishing our application a bit more.
Our site is, visually, almost complete. Let's quickly throw together a nicer looking "About Us" page, as that is the only page that we haven't touched yet.

To this section, we'll add a text section with some information about the application, and some information about the authentication system that we'll create in the next chapter.

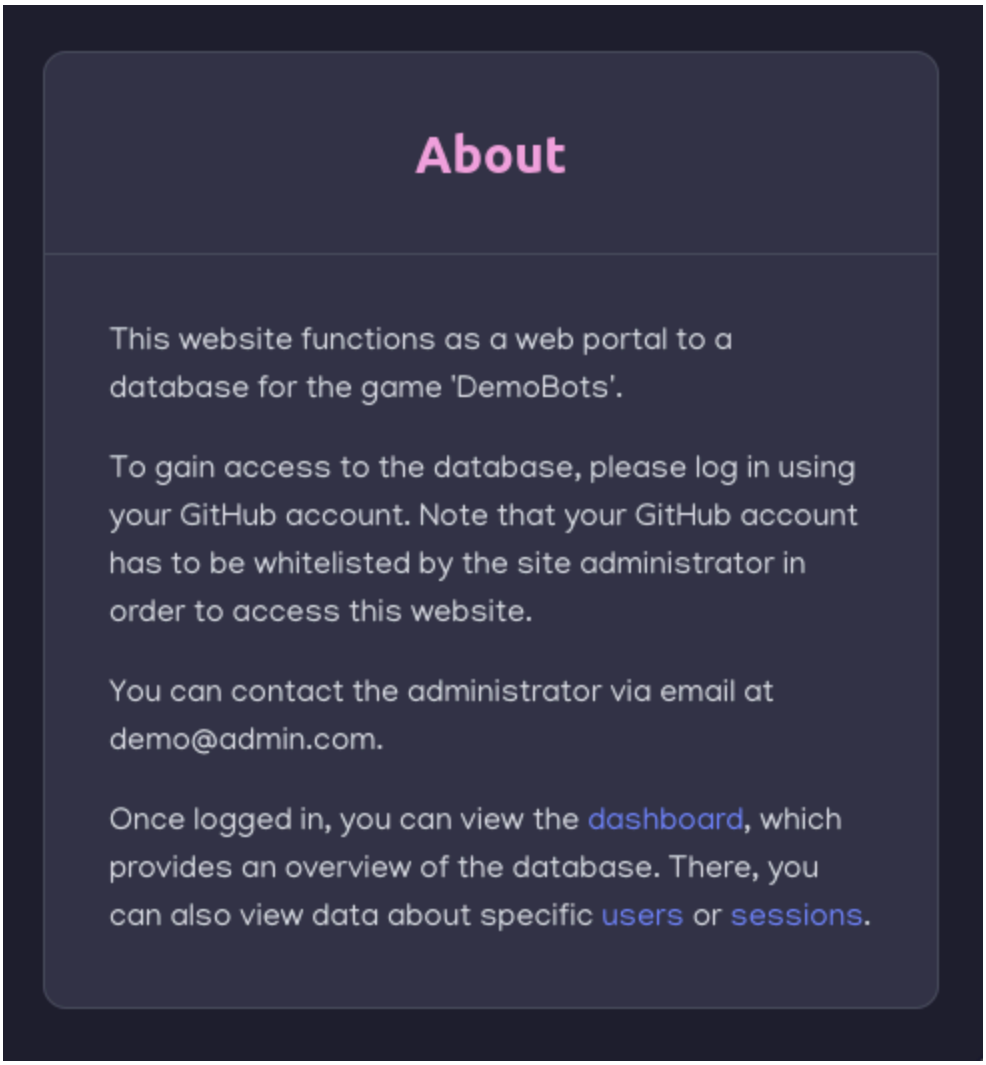
```
<!-- /src/routes/about/+page.svelte -->
<script lang="ts">
    import Card from '$lib/ui/views/Card.svelte';
    import { GAME_NAME, ADMIN_EMAIL } from '$lib/constants/strings';
</script>

<div class="flex h-full items-center justify-center p-4">
    <Card>
```

```
        {#snippet header()}
            <h1 class="text-center text-2xl font-bold">About</h1>
        {/snippet}
        {#snippet article()}
            <p>
                This website functions as a web portal to a database for the
game '{GAME_NAME}'.
            </p>
            <p>
                To gain access to the database, please log in using your
GitHub account. Note that your
                GitHub account has to be whitelisted by the site
administrator in order to access this
                website.
            </p>
            <p>
                You can contact the administrator via email at
{ADMIN_EMAIL}.
            </p>

            <p>
                Once logged in, you can view the <a href="/dashboard"
class="text-primary-500">dashboard</a
                >, which provides an overview of the database. There, you
can also view data about specific
                <a href="/dashboard/user" class="text-primary-500">users</a>
                or <a href="/dashboard/session" class="text-primary-
500">sessions</a>.
            </p>
        {/snippet}
    </Card>
</div>
```

We'll need to add the `ADMIN_EMAIL` to the `/src/lib/constants/strings.ts` file, so that we can use it in the `About` page:

```
// /src/lib/constants/strings.ts

...

export const ADMIN_EMAIL = 'demo@admin.com';
```

And now we have this nice card on our About page:



Wrapping up

In this chapter, we've nearly completed our application, loading our data from the database and displaying it in a user-friendly way.

We've learned how to:

- Load data into pages using the `load` function.
- Write clean, organized queries using DAOs (Data Access Objects).
- Handle route parameters and URL queries to load specific data.
- Display information using tables and charts that are both functional and visually clear.
- Make the app feel faster and more interactive with responsive search.
- Deal with errors gracefully, so users get helpful feedback when something goes wrong.

By now, we've connected the dots between our database and our UI — and made sure your users can actually see and interact with the data you've stored.

Next up, we'll go beyond reading data — and explore how we can build a secure authentication system to manage who can access our application.
Refer to Chapter 5: Authentication to learn how to implement authentication in your SvelteKit application.