

3 - Drizzle ORM

Now that we have a basic website up and running, we can start adding some data to it. In this chapter, we will set up [Drizzle](#) as our Object Relational Mapper (ORM) to interact with our MySQL database.

Chapter Overview

Learning goals

At the end of this chapter, you will:

- Understand what ORMs are and why they're beneficial for database interactions
- Configure Drizzle ORM in your SvelteKit project and set up MySQL database connections
- Define database schemas using Drizzle's TypeScript syntax
- Use Drizzle Kit to push schemas to your database and pull existing schemas
- Build complex relational database structures by implementing one-to-one and one-to-many relationships
- Generate realistic test data using drizzle-seed's built-in generators and refinement options
- Enhance seeded data with Faker.js for more sophisticated data generation scenarios
- Write basic queries to retrieve and manipulate data using Drizzle ORM

Learning resources

- [Drizzle ORM Official Documentation](#) - Comprehensive guide covering all features
- [Drizzle MySQL Column Types](#) - Reference for all available column types
- [Migrations with Drizzle Kit](#) - CLI tools for schema management
- [Drizzle Seed Documentation](#) - Guide to generating test data
- [Drizzle Seed Functions Reference](#) - Built-in data generators
- [Faker.js Documentation](#) - Library for generating realistic fake data
- [Database Relationships Explained](#) - Visual guide to database design
- [Foreign Keys vs Soft Relations](#) - Understanding different relationship approaches
- [Environment Variables in SvelteKit](#) - Secure configuration management

Introduction to ORMs

An ORM (Object Relational Mapper) is a programming technique that allows us to interact with a database using an object-oriented approach. Instead of writing raw SQL queries, we can use

an ORM to define our database tables and relationships as classes and objects in our code. This allows us to work with our data in a more intuitive way, and can help us avoid some of the common pitfalls of raw SQL queries, such as SQL injection attacks.

i

[Drizzle](#) is a TypeScript ORM that is designed to be simple and easy to use. It allows us to define our database tables and relationships using TypeScript classes, and provides a simple API for querying and manipulating our data. Drizzle also provides `drizzle-seed`, a tool for seeding our database with initial data. This can be useful for testing and development purposes, as it allows us to quickly populate our database with some sample data without having to write a bunch of SQL queries by hand.

It is a code-first ORM, meaning that we define our database schema in code, and then use the Drizzle CLI to push that schema to our database.

Setting up Drizzle ORM

During the setup of our SvelteKit project, we already installed `drizzle`, and the `sv` CLI has already generated the necessary configuration files for us. If you didn't install `drizzle` during the setup, you can do so now with the `sv cli`:


```
npx sv add drizzle
```

Or, if you wanted to, you could follow the [installation instructions](#) on the Drizzle website instead. We'll be covering some of the files that the `sv` CLI generated for us, so if you want to follow along, make sure to use the same configuration.

Drizzle configuration

As the `sv` CLI has already installed Drizzle into our project during setup, let's talk about some of the files it's created for us.


- `.env` and `.env.example`
 - These files are used to store environment variables. The `.env` file is used to store sensitive information, such as database connection strings, while the `.env.example` file is used to provide a template for the `.env` file, and leaves the values empty.

 **Important:** The `.env` file should **never** be committed to version control, as it contains sensitive information such as database credentials. The `.env.example` file is safe to commit, as it should not contain any sensitive information.

- `drizzle.config.ts`

- This file is used to configure Drizzle. It contains the database connection string and other configuration options. You can find more information about the configuration options in the [Drizzle documentation](#).
- `src/lib/server/db/index.ts`
 - This file is used to create a connection to the database and export the Drizzle instance. You can find more information about the configuration options in the [Drizzle documentation](#).
- `src/lib/server/db/schema.ts`
 - This file is used to define the database schema. It contains the database tables and relationships. You can find more information about the schema options in the [Drizzle documentation](#).

In `.env`, we want to update the `DATABASE_URL` to reflect the proper configuration. I'm using `root` (the default privileged MySQL user in XAMPP) as the `username` here (and optionally, a `password`), the `host` is `localhost`, on port `3306` (default port for the SQL database provided in XAMPP), and we point at a database `db-demobots` (that we have yet to create!). If you've messed around with the usernames, ports and/or passwords, fill those in instead.

 We won't be going live with this database, but it might be best practice to secure it using a password anyway. You can do so by running a query in the MySQL server (launch it by running `/opt/lampp/bin/mysql`): `SET PASSWORD FOR 'root'@'localhost' = PASSWORD('my_very_secretive_password');` `FLUSH PRIVILEGES;`. Be sure to update your `mysql` and `phpMyAdmin` credentials by editing `/opt/lampp/etc/my.cnf` (needs `sudo` access) and `/opt/lampp/phpmyadmin/config.inc.php` respectively. Don't forget to restart the XAMPP server (`sudo /opt/lampp/lampp restart`) afterwards. Should you ever run into issues, you can always attempt to [reset the password](#).

Following above instructions, your `.env` file should look something like this:

```
# Replace with your DB credentials!
DATABASE_URL="mysql://root:a_very_secretive_password@localhost:3306/db-demobots"
```

Your `/opt/lampp/etc/my.cnf` file should include something like this (note that this file is overridden by the alternative locations mentioned in the `my.cnf` file, `/opt/lampp/var/mysql/my.cnf` and `~/.my.cnf`):

```
# The following options will be passed to all MySQL clients
[client]
```

```
password      =my_very_secretive_password
port          =3306
```

And your `/opt/lampp/phpmyadmin/config.inc.php` file should include something like this:

```
/* Authentication type */
$cfg['Servers'][$i]['auth_type'] = 'config';
$cfg['Servers'][$i]['user'] = 'root';
$cfg['Servers'][$i]['password'] = 'my_very_secretive_password';
```

(Re)start the XAMPP server now to ensure the changes take effect:

```
sudo /opt/lampp/lampp restart
```

Next, let's take a look at the `src/lib/server/db/index.ts` file that was generated for us:

```
// /src/lib/server/db/index.ts

import { drizzle } from 'drizzle-orm/mysql2';
import mysql from 'mysql2/promise';
import * as schema from './schema';
import { env } from '$env/dynamic/private';

if (!env.DATABASE_URL) throw new Error('DATABASE_URL is not set');


const client = mysql.createPool(env.DATABASE_URL);

export const db = drizzle(client, { schema, mode: 'default' });
```

It's responsible for creating a connection to the MySQL database using the `mysql2` package, and exporting a Drizzle instance (`db`) that we can use to interact with our database. The `env` import is used to access the environment variables defined in our `.env` file.

We don't need to change anything in this file, but it's good to know what it does.

Creating the database

 If you already have an existing database that you want to use, you *can* skip this step, but it is highly recommended to create a new database entirely to avoid any data loss while we're figuring Drizzle out.

Now that we have our Drizzle configuration set up, before we start defining our database schema, we need to create the database itself.

This is as simple as visiting `[http://localhost/phpmyadmin]` (`http://localhost/phpmyadmin`) in your browser, which should log you in automatically using the credentials we provided in the `/opt/lampp/phpmyadmin/config.inc.php` file. Once at the landing page, click on the "Databases" tab at the top, and then in the "Create database" field, enter `db-demobots` (or whatever name you set up earlier in your `.env` file), and click the "Create" button.

We won't be doing anything else in phpMyAdmin for now, as we will be using Drizzle (`drizzle-seed` to be exact) to set up our database schema and seed our database with some test data.

Defining the database schema

i If you already have an existing database with a schema that you want to use, you can skip this step and instead skip forward to the sub-section on "Pulling a schema from an existing database". However, once again, it is highly recommended to just follow along with the steps we're taking here to avoid any data loss while we're figuring Drizzle out.

Exploring the schema

Let's take a look at the `src/lib/server/db/schema.ts` file that was generated for us:

```
// /src/lib/db/schema.ts

import { mysqlTable, serial, int } from 'drizzle-orm/mysql-core';

export const user = mysqlTable('user', {
  id: serial('id').primaryKey(),
  age: int('age')
});
```

This file currently defines a single table called `user`, with two columns: `id` and `age`. The `id` column is a serial (auto-incrementing) primary key, and the `age` column is an integer. It imports the necessary functions from the `drizzle-orm/mysql-core` package to define the type of table and column types.

i Even though present in the default schema, the SQL Database we've installed along with XAMPP (MariaDB) does not support the `serial` type directly. We will change this in the next step, but keep this in mind, as you might run into similar column type errors pushing your schema to the database.

Altering the schema

Users Table

Let's expand our database with a simple user table that contains some more information about the users. Looking at the [column types](#) that Drizzle supports, let's create a `user` table that contains the following columns:

Column Name	Data Type	Constraints
username	varchar(20)	UNIQUE, NOT NULL
id	int	PRIMARY KEY, AUTO_INCREMENT
created_at	datetime	NOT NULL
date_of_birth	date	NOT NULL

Implementation

We'll add the following columns to the `user` table.

```
// /src/lib/db/schema.ts

import { mysqlTable, date, datetime, varchar, int } from 'drizzle-orm/mysql-core';

// Holds user information.
export const userTable = mysqlTable('user', {
  id: int('id').primaryKey().autoincrement(),
  createdAt: datetime('created_at').notNull(),
  username: varchar('username', { length: 20 }).notNull(),
  dateOfBirth: date('date_of_birth').notNull()
});
```

Now we want to have Drizzle create this table in our database.

So let's try pushing this schema to our database.

Pushing a schema

To do this, we can use the `drizzle-kit` CLI tool that was installed as part of the Drizzle setup. We can run it with `npx`:

[Pushing a schema](#) to the database is the act of altering the database to match the schema defined in our code.

```
npx drizzle-kit push
```

We will be met with a message showing the SQL statement that was generated to add these changes to our database. Confirming that this is what we want to do, the statement will be executed, and the table will be created in our database:

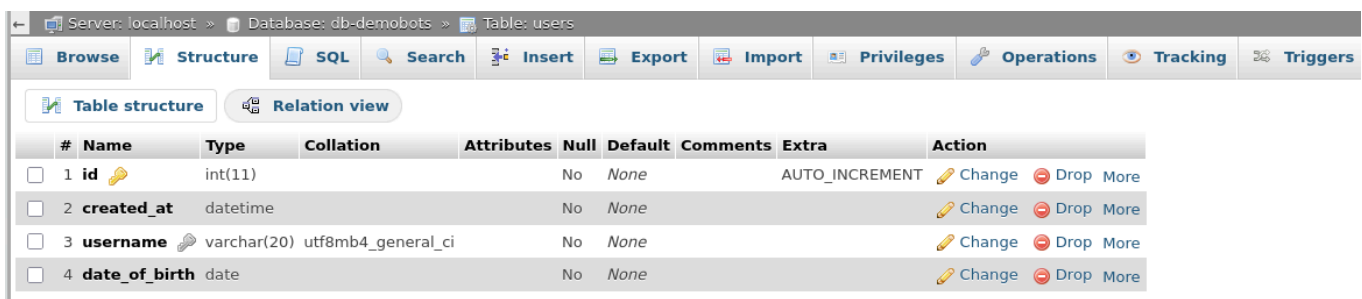
```
🏠 /app on main ↑3 🔗 ?7 ~3
> npx drizzle-kit push
No config path provided, using default 'drizzle.config.ts'
Reading config file '/home/codevotel/work/svelte-db-portal/app/drizzle.config.ts'
[✓] Pulling schema from database...
Reading schema files:
/home/codevotel/work/svelte-db-portal/app/src/lib/server/db/schema.ts

Warning You are about to execute current statements:

CREATE TABLE `users` (
  `id` int AUTO_INCREMENT NOT NULL,
  `created_at` datetime NOT NULL,
  `username` varchar(20) NOT NULL,
  `date_of_birth` date NOT NULL,
  CONSTRAINT `users_id` PRIMARY KEY(`id`),
  CONSTRAINT `users_username_unique` UNIQUE(`username`)
);
```

⚠ If you push to a database that already has tables, Drizzle will attempt to *alter* the existing tables to match the schema defined in your code. This can however cause some issues with data loss. For simplicity during this course, just *drop* all the tables in your dummy database before pushing the schema, so that Drizzle can create the tables from scratch, and then seed the database (see next section) to repopulate the tables with data.

Let's go over to <http://localhost/phpmyadmin/>, open up the `db-demobots` database, and then click on the `user` table to see the structure of the table we just created:



Server: localhost » Database: db-demobots » Table: users									
Browse Structure SQL Search Insert Export Import Privileges Operations Tracking Triggers									
Table structure Relation view									
#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra	Action
<input type="checkbox"/> 1	id	int(11)			No	None		AUTO_INCREMENT	Change Drop More
<input type="checkbox"/> 2	created_at	datetime			No	None			Change Drop More
<input type="checkbox"/> 3	username	varchar(20)	utf8mb4_general_ci		No	None			Change Drop More
<input type="checkbox"/> 4	date_of_birth	date			No	None			Change Drop More

Pulling a schema

Looks good! We'll create some more interesting tables that share relations later, but for now, let's try and do the reverse: let's pull a schema from an existing database instead.

[Pulling a schema](#) is the opposite of pushing a schema. It takes a look at an existing database and generates a Drizzle schema file that matches the structure of the database.

If you already have an existing database with a schema that you want to use, you can use `drizzle-kit` to pull the schema from the database and generate the corresponding Drizzle schema file.

Let's do this by running the following command:

```
npx drizzle-kit pull
```

As seen below, the CLI pulls the schema from the database, and spits out a `drizzle/schema.ts` file and `drizzle/relations.ts` file (we'll get into relations later) that, when pushed, would recreate the database structure.

```
🏠/📁/app on 🌀 main ≡ 🔗 ?2 ~4
> npx drizzle-kit pull
No config path provided, using default 'drizzle.config.ts'
Reading config file '/home/codevangel/work/svelte-db-portal/app/drizzle.config.ts'
Pulling from ['public'] list of schemas

[✓] 1 tables fetched
[✓] 6 columns fetched
[✓] 0 indexes fetched
[✓] 0 foreign keys fetched
[::] 0 policies fetching
[✓] 0 check constraints fetched
[✓] 0 views fetched

[✓] Your SQL migration file → drizzle/0000\_cooing\_slyde.sql 🚀
[✓] Your schema file is ready → drizzle/schema.ts 🚀
[✓] Your relations file is ready → drizzle/relations.ts 🚀
```

We won't be using this feature too much, so you can go ahead and delete the created `drizzle` folder, but it's neat to know about.

i If you had an existing database with a schema that you wanted to use, it is still a good recommendation to create a new dummy database and run through the steps of defining the schema from scratch, as it will help you understand how Drizzle works and how to define your own schema in the future, and since we'll be dropping tables instead of migrating between changes, you will lose data in your real database.

Now that we know how to push and pull our schema, let's move on to seeding our database with some initial data.

Seeding the database

To be able to test our application, we need some initial data in our database. Now we *could* painstakingly write SQL queries to manually insert data into our database, or use PHPMyAdmin to do so, but Drizzle offers a much simpler way to seed our database with some realistic data using `drizzle-seed`.

`drizzle-seed` [\(link\)](#) is a TypeScript library that helps you generate deterministic, yet realistic, fake data to populate your database.

To use `drizzle-seed`, we need to install a few dependencies;

- `drizzle-seed` itself, which provides the seeding functionality.
- `dotenv`, which allows us to load environment variables from a `.env` file.
- `tsx`, which is a TypeScript execution environment that allows us to run TypeScript files directly.

We can install these dependencies by running the following command in our terminal:

```
npm install --save-dev drizzle-seed dotenv tsx
```

Now that we have the necessary dependencies installed, we can create a seeding script to populate our database with some initial data. Let's create a new file called `reseed.ts` in a new directory `scripts` which we'll place in the root directory of our svelte project (`/scripts/reseed.ts`):

```
// /scripts/reseed.ts

import mysql from 'mysql2/promise';
// This import is necessary so we can load our environment variables
import 'dotenv/config';
// This should point to your schema file
import * as schema from '../src/lib/server/db/schema';
import { drizzle } from 'drizzle-orm/mysql2';
import { reset, seed } from 'drizzle-seed';

// Function to reseed the database
async function reseed_db() {
  // Ensure the DATABASE_URL environment variable is set
  if (!process.env.DATABASE_URL) throw new Error('DATABASE_URL is not set');

  // Create a MySQL client and initialize Drizzle ORM
```

```

const client = mysql.createPool(process.env.DATABASE_URL);
const db = drizzle(client, { schema, mode: 'default' });

// Reset the database (drop all data from the tables)
console.log('Resetting database...');
await reset(db, schema);
// Seed the database with test data
console.log('Reseeding database...');
await seed(db, schema);

// Log success message
console.log('Database reseeded successfully');
await client.end();
}

// Entry point for the script
async function main() {
  await reseed_db();
}

// Run the main function
main();

```

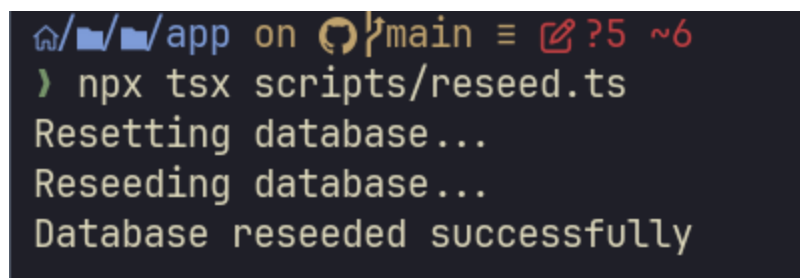
If we run this script, our database will be reset (all data will be lost(!)) and then (re)seeded with some initial data.

This data is generated using a pseudo-random number generator, meaning that the data will be random, but with a deterministic seed, so that we can generate the same data every time we run the script. (You can see that that becomes very useful when testing things later on.)

To run the script, we can use the `tsx` command, which allows us to run TypeScript files directly:

```
npx tsx scripts/reseed.ts
```

If everything goes well, we should see our message logs indicating that the database has been reset and reseeded successfully:

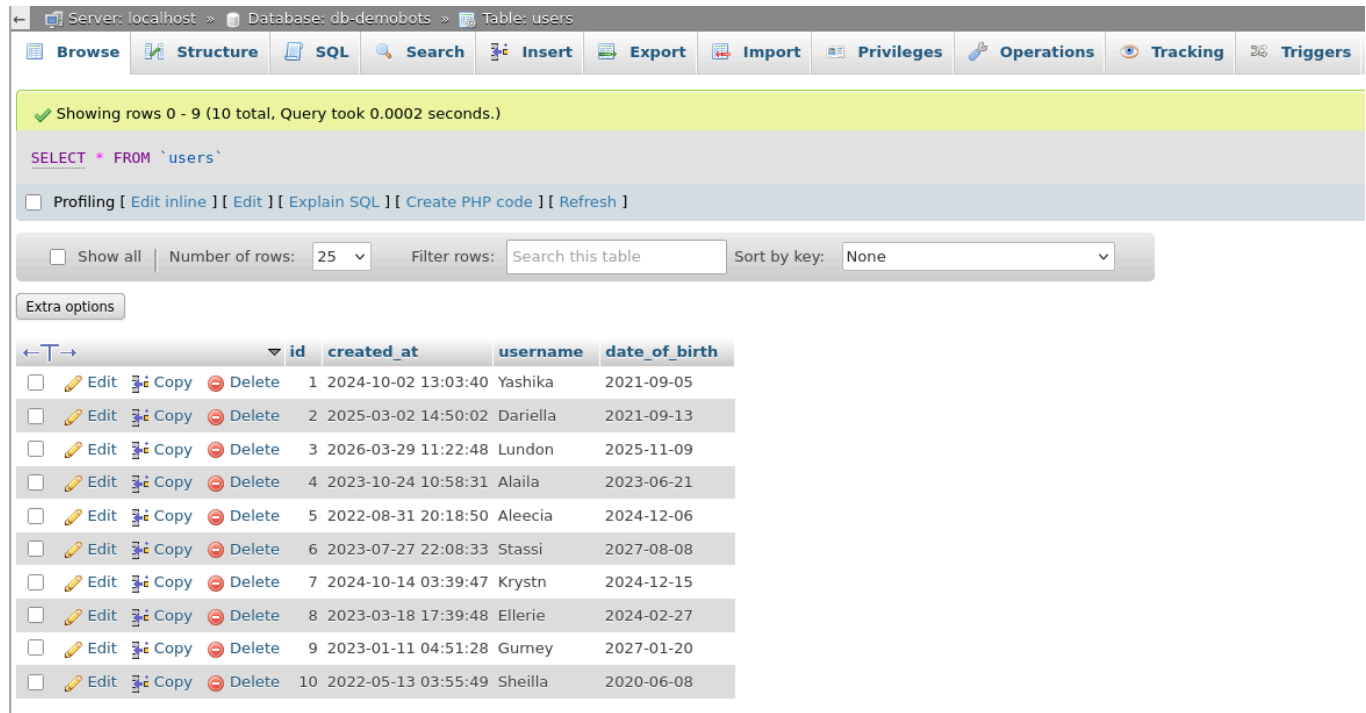


```

🏠/📁/📁/app on 🔄 main ≡ 🔗 ?5 ~6
> npx tsx scripts/reseed.ts
Resetting database...
Reseeding database...
Database reseeded successfully

```

Let's go take a look at our table in `phpMyAdmin` again, so we can see the newly generated data:



The screenshot shows the phpMyAdmin interface for a database named 'db-demobots' on 'localhost'. The 'Table: users' is selected. The interface includes tabs for Browse, Structure, SQL, Search, Insert, Export, Import, Privileges, Operations, Tracking, and Triggers. A status bar indicates 'Showing rows 0 - 9 (10 total, Query took 0.0002 seconds.)'. Below this, a SQL query 'SELECT * FROM `users`' is shown. A toolbar contains links for Profiling, Edit inline, Edit, Explain SQL, Create PHP code, and Refresh. A filter section shows 'Show all', 'Number of rows: 25', 'Filter rows: Search this table', and 'Sort by key: None'. An 'Extra options' button is also present. The table data is displayed with columns: id, created_at, username, and date_of_birth. Each row has links for Edit, Copy, and Delete.

	id	created_at	username	date_of_birth
<input type="checkbox"/> Edit Copy Delete	1	2024-10-02 13:03:40	Yashika	2021-09-05
<input type="checkbox"/> Edit Copy Delete	2	2025-03-02 14:50:02	Dariella	2021-09-13
<input type="checkbox"/> Edit Copy Delete	3	2026-03-29 11:22:48	London	2025-11-09
<input type="checkbox"/> Edit Copy Delete	4	2023-10-24 10:58:31	Alaila	2023-06-21
<input type="checkbox"/> Edit Copy Delete	5	2022-08-31 20:18:50	Aleecia	2024-12-06
<input type="checkbox"/> Edit Copy Delete	6	2023-07-27 22:08:33	Stassi	2027-08-08
<input type="checkbox"/> Edit Copy Delete	7	2024-10-14 03:39:47	Krystn	2024-12-15
<input type="checkbox"/> Edit Copy Delete	8	2023-03-18 17:39:48	Ellerie	2024-02-27
<input type="checkbox"/> Edit Copy Delete	9	2023-01-11 04:51:28	Gurney	2027-01-20
<input type="checkbox"/> Edit Copy Delete	10	2022-05-13 03:55:49	Sheilla	2020-06-08

Cool. This will save us a lot of time doing data-entry by hand, and ensures that we make no mistakes in the data we enter.

i Just as a heads-up, we can (and will) later refine this data to adhere to specific testing needs (such as the date of births all being in a certain range, or adding special characters to usernames).

Tables that share relations

Remember, an ORM stands for Object *Relational* Mapper, but right now, we only really have a single table with some Objects in it. We can do better than that: we can create tables that share relations with each other, allowing us to model more complex data structures.

Soft Relations vs Foreign Keys

Before we dive into implementing relations in Drizzle, let's clarify the difference between **relations** and **foreign keys**.

- **Foreign keys** are a database-level constraint that allows us to link two tables together. A foreign key is a column in one table that references the primary key of another table. This creates a relationship between the two tables, allowing us to join them together in queries. Foreign key constraints make sure that when inserting/updating/deleting data, the data remains consistent and valid across the tables. For example, if we have a `user_profiles`

table that references the `users` table, we can ensure that every profile belongs to a valid user.

- **Soft Relations** on the other hand, are an application-level abstraction that Drizzle provides to make it easier to work with related data. Crucial to know is that they do not affect the database schema directly, but they can be very useful for making queries more readable and easier to work with.
- We can define soft relations without foreign keys and vice versa, but in most cases, it is a good idea to use both.

It's okay to be a bit confused by this at first, we just wanted to clarify the difference between the two before we start thinking of them as interchangeable.

For now, let's just take a look at a simple relationship between two tables: the One-to-One relationship.

One-to-one

[One to One relationships](#) are the simplest type of relationship, where one record in a table is related to one record in another table. For example, we could have a `profile` table that contains additional information about a user, rather than storing less crucial data in the `user` table itself.

Let's set up a `user_profiles` table that holds additional information about the user, with the following format:

User Profiles Table

Column Name	Data Type	Constraints
<code>id</code>	<code>int</code>	PRIMARY KEY, AUTO_INCREMENT
<code>user_id</code>	<code>int</code>	FOREIGN KEY → <code>users(id)</code> , NOT NULL
<code>first_name</code>	<code>varchar(35)</code>	NOT NULL
<code>last_name</code>	<code>varchar(35)</code>	NOT NULL
<code>title</code>	<code>varchar(35)</code>	NOT NULL

Implementation

```
// /src/lib/db/schema.ts

import { mysqlTable, date, datetime, varchar, int } from 'drizzle-orm/mysql-core';
import { relations } from 'drizzle-orm';
```

```

// Holds user information.
export const users = mysqlTable('users', {
  id: int('id').primaryKey().autoincrement(),
  createdAt: datetime('created_at').notNull(),
  username: varchar('username', { length: 20 }).unique().notNull(),
  dateOfBirth: date('date_of_birth').notNull()
});

// Holds user profile information.
export const userProfiles = mysqlTable('user_profiles', {
  id: int('id').primaryKey().autoincrement(),
  userId: int('user_id').references(() => users.id).notNull(),
  firstName: varchar('first_name', { length: 35 }).notNull(),
  lastName: varchar('last_name', { length: 35 }).notNull(),
  title: varchar('title', { length: 35 }).notNull()
});

// One user has one profile.
export const usersRelations = relations(users, ({ one }) => ({
  profile: one(userProfiles)
})));

// One profile belongs to one user.
export const userProfilesRelations = relations(userProfiles, ({ one }) => ({
  user: one(users, {
    fields: [userProfiles.userId],
    references: [users.id]
  })
})));


```

Now we have two tables: `users` and `user_profiles`. One user can have one profile, and one profile belongs to one user.

- The `users` table now leaves out the `firstName` and `lastName` columns.
- The `user_profiles` table is new, and contains:
 - `id`: an auto-incrementing primary key of type `int`.
 - `userId`: a foreign key that references the `id` column in the `users` table. More on foreign keys later.
 - `firstName`: a `varchar` column that stores the user's first name, with a maximum length of 35 characters.
 - `lastName`: a `varchar` column that stores the user's last name, with a maximum length of 35 characters.

- `title`: a `varchar` column that stores the user's title (e.g. their job title), with a maximum length of 35 characters.

We also define the **relations** between the two tables using the `relations` function from Drizzle. This allows us to easily query the related data later on.

Let's drop the existing `user` tables in our database using PHPMysqlAdmin (click the  icon in the table overview), and then push our schema again with `npx drizzle-kit push`. We can see how the SQL statement generated by Drizzle automatically adds the constraints that we defined in our schema:

```
~/app on main ↑3 ?2 ~3
> npx drizzle-kit push
No config path provided, using default 'drizzle.config.ts'
Reading config file '/home/codevogel/work/svelte-db-portal/app/drizzle.config.ts'
[✓] Pulling schema from database...
Reading schema files:
/home/codevogel/work/svelte-db-portal/app/src/lib/server/db/schema.ts

Warning You are about to execute current statements:

CREATE TABLE `user_profiles` (
  `id` int AUTO_INCREMENT NOT NULL,
  `user_id` int,
  `first_name` varchar(30) NOT NULL,
  `last_name` varchar(30) NOT NULL,
  `title` varchar(255) NOT NULL,
  CONSTRAINT `user_profiles_id` PRIMARY KEY(`id`)
);

CREATE TABLE `users` (
  `id` int AUTO_INCREMENT NOT NULL,
  `created_at` datetime NOT NULL,
  `username` varchar(20) NOT NULL,
  `date_of_birth` date NOT NULL,
  CONSTRAINT `users_id` PRIMARY KEY(`id`),
  CONSTRAINT `users_username_unique` UNIQUE(`username`)
);

ALTER TABLE `user_profiles` ADD CONSTRAINT `user_profiles_user_id_users_id_fk` FOREIGN KEY
(`user_id`) REFERENCES `users`(`id`) ON DELETE no action ON UPDATE no action;

[✓] Changes applied
```

One-to-many

[One to Many relationships](#) are a type of relationship where one record in a table is related to multiple records in another table. For example, we could have a `sessions` table that contains data about the user's training sessions in our game, where one user can have multiple sessions, but each session belongs to one user.

Sessions table

Let's set up a `sessions` table that holds information about the user's training session, with the following format:

Column Name	Data Type	Constraints
<code>id</code>	<code>int</code>	PRIMARY KEY, AUTO_INCREMENT
<code>user_id</code>	<code>int</code>	FOREIGN KEY → <code>users(id)</code> , NOT NULL
<code>created_at</code>	<code>datetime</code>	NOT NULL
<code>duration</code>	<code>int</code>	NOT NULL

Implementation

We'll implement it as follows:

```
// Holds information about user sessions
export const sessions = mysqlTable('sessions', {
  id: int('id').primaryKey().autoincrement(),
  userId: int('user_id').references(() => users.id).notNull(),
  createdAt: datetime('created_at').notNull(),
  duration: int('duration').notNull() // in seconds
});

// Define relations from the sessions table
export const sessionsRelations = relations(sessions, ({ one }) => ({
  // One session belongs to one user.
  user: one(users, {
    fields: [sessions.userId],
    references: [users.id]
  })
}));
```

Finishing up the schema

Now that we understand how to create new tables and define relations between them, let's finish up our schema by adding a few more tables that will help us model the data for our serious game database.

Score Table

This table keeps track of the scores that a user achieves. Each score is associated with a specific training session, and is achieved on a specific level of the game.

This table will hold the following columns:

Column Name	Data Type	Constraints
id	int	PRIMARY KEY, AUTO_INCREMENT
session_id	int	FOREIGN KEY → sessions(id), NOT NULL
level_id	int	FOREIGN KEY → levels(id), NOT NULL

Implementation

We will implement the `scores` table as follows:

```
// Holds scores for each session and level
export const scores = mysqlTable('scores', {
  id: int('id').primaryKey().autoincrement(),
  sessionId: int('session_id').references(() => sessions.id).notNull(),
  levelId: int('level_id').references(() => levels.id).notNull(),
  createdAt: datetime('created_at').notNull(),
  score: int('score').notNull(),
  timeTaken: int('time_taken').notNull(),
  accuracy: float('accuracy').notNull()
});
```

And we update the `sessionsRelations` and add the `scoresRelations`:

```
// Define relations from the sessions table
export const sessionsRelations = relations(sessions, ({ one, many }) => ({
  ...,
  scores: many(scores) // One session can have many scores.
}));

// Define relations from the score table
export const scoresRelations = relations(scores, ({ one }) => ({
  // One score belongs to one session.
  session: one(sessions, {
    fields: [scores.sessionId],
    references: [sessions.id]
  })
}));
```

Level Table

This table holds information about the different levels in the game. Each level has a unique ID and a name.

This table will hold the following columns:

Column Name	Data Type	Constraints
id	int	PRIMARY KEY, AUTO_INCREMENT
name	varchar(50)	NOT NULL
difficulty	int	NOT NULL

Implementation

We will implement the `levels` table as follows:

```
// Holds information about the levels in the game
export const levels = mysqlTable('levels', {
  id: int('id').primaryKey().autoincrement(),
  name: varchar('name', { length: 50 }).notNull(),
  difficulty: int('difficulty').notNull()
});
```

And we'll update the `scoresRelations` and add the `levelsRelations`:

```
// Define relations from the score table
export const scoresRelations = relations(scores, ({ one }) => ({
  ...
  // One score belongs to one level.
  level: one(levels, {
    fields: [scores.levelId],
    references: [levels.id]
  })
}));

// Holds information about the levels in the game
export const levelsRelations = relations(levels, ({ many }) => ({
  // One level can have many scores.
  scores: many(scores)
}));
```

Final Schema

The final (for now) schema we ended up with during this modifications should look like this:

```
// /src/lib/db/schema.ts

import { mysqlTable, date, datetime, varchar, int, float } from 'drizzle-orm/mysql-core';
import { relations } from 'drizzle-orm';

// Holds user information.
export const users = mysqlTable('users', {
  id: int('id').primaryKey().autoincrement(),
  createdAt: datetime('created_at').notNull(),
  username: varchar('username', { length: 20 }).unique().notNull(),
  dateOfBirth: date('date_of_birth').notNull()
});

// Holds user profile information.
export const userProfiles = mysqlTable('user_profiles', {
  id: int('id').primaryKey().autoincrement(),
  userId: int('user_id').references(() => users.id).notNull(),
  firstName: varchar('first_name', { length: 35 }).notNull(),
  lastName: varchar('last_name', { length: 35 }).notNull(),
  title: varchar('title', { length: 35 }).notNull()
});

// Holds information about user sessions
export const sessions = mysqlTable('sessions', {
  id: int('id').primaryKey().autoincrement(),
  userId: int('user_id').references(() => users.id).notNull(),
  createdAt: datetime('created_at').notNull(),
  duration: int('duration').notNull() // in seconds
});

// Holds scores for each session and level
export const scores = mysqlTable('scores', {
  id: int('id').primaryKey().autoincrement(),
  sessionId: int('session_id').references(() => sessions.id).notNull(),
  levelId: int('level_id').references(() => levels.id).notNull(),
  createdAt: datetime('created_at').notNull(),
  score: int('score').notNull(),
  timeTaken: int('time_taken').notNull(),
  accuracy: float('accuracy').notNull()
});

export const levels = mysqlTable('levels', {
  id: int('id').primaryKey().autoincrement(),
```

```

    name: varchar('name', { length: 50 }).notNull(),
    difficulty: int('difficulty').notNull()
  });

// Define relations from the users table
export const usersRelations = relations(users, ({ one, many }) => ({
  // One user has one profile.
  profile: one(userProfiles),
  // One user can have many sessions.
  sessions: many(sessions)
}));

// Define relations from the userProfiles table
export const userProfilesRelations = relations(userProfiles, ({ one }) => ({
  // One user profile belongs to one user.
  user: one(users, {
    fields: [userProfiles.userId],
    references: [users.id]
  })
}));

// Define relations from the sessions table
export const sessionsRelations = relations(sessions, ({ one, many }) => ({
  // One session belongs to one user.
  user: one(users, {
    fields: [sessions.userId],
    references: [users.id]
  }),
  scores: many(scores) // One session can have many scores.
}));

// Define relations from the score table
export const scoresRelations = relations(scores, ({ one }) => ({
  // One score belongs to one session.
  session: one(sessions, {
    fields: [scores.sessionId],
    references: [sessions.id]
  }),
  // One score belongs to one level.
  level: one(levels, {
    fields: [scores.levelId],
    references: [levels.id]
  })
}));

// Holds information about the levels in the game

```

```
export const levelsRelations = relations(levels, ({ many }) => ({
  // One level can have many scores.
  scores: many(scores)
}));
```

Seeding refinements

Now that we have a more complex schema, where tables actually share relations, we should update our seeding script to generate more realistic data that adheres to the schema we've defined.

i Please do follow along with the code snippets below, as they will help you understand how to refine the generated data to better suit your needs. However, keep in mind that you should not overly complicate the seeding script for your own database. Just focus on the most important aspects of the data that need to be 'realistic' for your application. In our case, we want to show some realistic data in graphs, so we need to ensure that the dates, durations, and scores are all within a reasonable range.

Relation-aware seeding

Drizzle's `drizzle-seed` library handles relations pretty well. We just need to give it some hints on how to generate the data.

We can do so by using the `refine` method.

Let's update our `scripts/reseed.ts` file to use the `refine` method:

```
// /scripts/reseed.ts
...

await seed(db, schema).refine((f) => ({
  users: {
    count: 12,
    with: {
      userProfiles: 1,
      sessions: 5
    }
  },
  sessions: {
    with: {
      scores: 5
    }
  },
  levels: {
    count: 5
```

```

    }
  }));

  ...

```

This will generate 12 users, each with 1 profile and 5 sessions.

Each session will have five scores, and the `levels` table will be populated with five levels.

Because we pass the `schema` to the `seed` function, Drizzle is aware of the relations between our tables, and will generate the data accordingly.

Thusly, we will end up with a database that has:

- 12 users
 - Each user has 1 profiles
 - Each user has 5 sessions
- 12 user profiles
- 60 sessions
 - Each session has 5 scores
- 300 scores
- 5 levels

Let's check it out in `phpMyAdmin` :

Table	Action	Rows	Type	Collation	Size	Overhead
<input type="checkbox"/> levels	★ Browse Structure Search Insert Empty Drop	5	InnoDB	utf8mb4_general_ci	16.0 KiB	-
<input type="checkbox"/> scores	★ Browse Structure Search Insert Empty Drop	300	InnoDB	utf8mb4_general_ci	48.0 KiB	-
<input type="checkbox"/> sessions	★ Browse Structure Search Insert Empty Drop	60	InnoDB	utf8mb4_general_ci	32.0 KiB	-
<input type="checkbox"/> users	★ Browse Structure Search Insert Empty Drop	12	InnoDB	utf8mb4_general_ci	32.0 KiB	-
<input type="checkbox"/> user_profiles	★ Browse Structure Search Insert Empty Drop	12	InnoDB	utf8mb4_general_ci	32.0 KiB	-
5 tables	Sum	389	InnoDB	utf8mb4_general_ci	160.0 KiB	0 B

Those row counts look right!

Looking at the `sessions` table, we can also see each `user_id` shows up five times, and each `user_id` is valid:

✓ Showing rows 0 - 24 (60 total, Query took 0.0001 seconds.) [user_id: 1... - 5...]

```
SELECT * FROM `sessions` ORDER BY `user_id` ASC
```

☐ Profiling [[Edit inline](#)] [[Edit](#)] [[Explain SQL](#)] [[Create PHP code](#)] [[Refresh](#)]

1 ▾

>

>>

☐

Show all

Number of rows:

25 ▾

Filter rows:

Extra options

<div><div><div></div><div></div><div></div></div></div>				id	user_id	1	created_at	duration		
<div><div><div></div><div></div><div></div></div></div>	<div><div><div></div><div></div><div></div></div></div>	Edit	<div><div><div></div><div></div><div></div></div></div>	Copy	<div><div><div></div><div></div><div></div></div></div>	Delete	54	1	2022-05-18 16:27:41	542229055
<div><div><div></div><div></div><div></div></div></div>	<div><div><div></div><div></div><div></div></div></div>	Edit	<div><div><div></div><div></div><div></div></div></div>	Copy	<div><div><div></div><div></div><div></div></div></div>	Delete	34	1	2023-07-17 02:08:57	-564113392
<div><div><div></div><div></div><div></div></div></div>	<div><div><div></div><div></div><div></div></div></div>	Edit	<div><div><div></div><div></div><div></div></div></div>	Copy	<div><div><div></div><div></div><div></div></div></div>	Delete	36	1	2022-07-25 15:54:34	62604760
<div><div><div></div><div></div><div></div></div></div>	<div><div><div></div><div></div><div></div></div></div>	Edit	<div><div><div></div><div></div><div></div></div></div>	Copy	<div><div><div></div><div></div><div></div></div></div>	Delete	58	1	2023-12-19 13:32:59	-1380656345
<div><div><div></div><div></div><div></div></div></div>	<div><div><div></div><div></div><div></div></div></div>	Edit	<div><div><div></div><div></div><div></div></div></div>	Copy	<div><div><div></div><div></div><div></div></div></div>	Delete	28	1	2022-06-14 03:24:58	-2105192343
<div><div><div></div><div></div><div></div></div></div>	<div><div><div></div><div></div><div></div></div></div>	Edit	<div><div><div></div><div></div><div></div></div></div>	Copy	<div><div><div></div><div></div><div></div></div></div>	Delete	27	2	2025-05-17 19:30:09	1451054108
<div><div><div></div><div></div><div></div></div></div>	<div><div><div></div><div></div><div></div></div></div>	Edit	<div><div><div></div><div></div><div></div></div></div>	Copy	<div><div><div></div><div></div><div></div></div></div>	Delete	19	2	2023-02-08 04:29:28	56664793
<div><div><div></div><div></div><div></div></div></div>	<div><div><div></div><div></div><div></div></div></div>	Edit	<div><div><div></div><div></div><div></div></div></div>	Copy	<div><div><div></div><div></div><div></div></div></div>	Delete	20	2	2024-02-21 10:43:34	-1022833800
<div><div><div></div><div></div><div></div></div></div>	<div><div><div></div><div></div><div></div></div></div>	Edit	<div><div><div></div><div></div><div></div></div></div>	Copy	<div><div><div></div><div></div><div></div></div></div>	Delete	21	2	2023-03-28 14:25:13	410637712
<div><div><div></div><div></div><div></div></div></div>	<div><div><div></div><div></div><div></div></div></div>	Edit	<div><div><div></div><div></div><div></div></div></div>	Copy	<div><div><div></div><div></div><div></div></div></div>	Delete	22	2	2025-09-16 04:56:31	1868628430
<div><div><div></div><div></div><div></div></div></div>	<div><div><div></div><div></div><div></div></div></div>	Edit	<div><div><div></div><div></div><div></div></div></div>	Copy	<div><div><div></div><div></div><div></div></div></div>	Delete	6	3	2022-06-12 06:47:50	1484626996
<div><div><div></div><div></div><div></div></div></div>	<div><div><div></div><div></div><div></div></div></div>	Edit	<div><div><div></div><div></div><div></div></div></div>	Copy	<div><div><div></div><div></div><div></div></div></div>	Delete	25	3	2023-09-20 01:42:08	1898467943
<div><div><div></div><div></div><div></div></div></div>	<div><div><div></div><div></div><div></div></div></div>	Edit	<div><div><div></div><div></div><div></div></div></div>	Copy	<div><div><div></div><div></div><div></div></div></div>	Delete	52	3	2023-12-31 10:30:11	1643402007
<div><div><div></div><div></div><div></div></div></div>	<div><div><div></div><div></div><div></div></div></div>	Edit	<div><div><div></div><div></div><div></div></div></div>	Copy	<div><div><div></div><div></div><div></div></div></div>	Delete	17	3	2023-09-21 00:46:35	-1731483424
<div><div><div></div><div></div><div></div></div></div>	<div><div><div></div><div></div><div></div></div></div>	Edit	<div><div><div></div><div></div><div></div></div></div>	Copy	<div><div><div></div><div></div><div></div></div></div>	Delete	1	3	2023-02-10 18:03:54	798080813
<div><div><div></div><div></div><div></div></div></div>	<div><div><div></div><div></div><div></div></div></div>	Edit	<div><div><div></div><div></div><div></div></div></div>	Copy	<div><div><div></div><div></div><div></div></div></div>	Delete	35	4	2026-04-08 00:00:59	392688368
<div><div><div></div><div></div><div></div></div></div>	<div><div><div></div><div></div><div></div></div></div>	Edit	<div><div><div></div><div></div><div></div></div></div>	Copy	<div><div><div></div><div></div><div></div></div></div>	Delete	47	4	2023-04-25 04:32:21	-1033017290
<div><div><div></div><div></div><div></div></div></div>	<div><div><div></div><div></div><div></div></div></div>	Edit	<div><div><div></div><div></div><div></div></div></div>	Copy	<div><div><div></div><div></div><div></div></div></div>	Delete	40	4	2023-07-12 21:57:45	-814098377
<div><div><div></div><div></div><div></div></div></div>	<div><div><div></div><div></div><div></div></div></div>	Edit	<div><div><div></div><div></div><div></div></div></div>	Copy	<div><div><div></div><div></div><div></div></div></div>	Delete	50	4	2022-10-10 07:07:53	2128906724
<div><div><div></div><div></div><div></div></div></div>	<div><div><div></div><div></div><div></div></div></div>	Edit	<div><div><div></div><div></div><div></div></div></div>	Copy	<div><div><div></div><div></div><div></div></div></div>	Delete	43	4	2023-03-29 02:10:01	1960946312
<div><div><div></div><div></div><div></div></div></div>	<div><div><div></div><div></div><div></div></div></div>	Edit	<div><div><div></div><div></div><div></div></div></div>	Copy	<div><div><div></div><div></div><div></div></div></div>	Delete	3	5	2024-04-17 05:53:57	-318190785
<div><div><div></div><div></div><div></div></div></div>	<div><div><div></div><div></div><div></div></div></div>	Edit	<div><div><div></div><div></div><div></div></div></div>	Copy	<div><div><div></div><div></div><div></div></div></div>	Delete	44	5	2023-04-18 03:59:35	663267890
<div><div><div></div><div></div><div></div></div></div>	<div><div><div></div><div></div><div></div></div></div>	Edit	<div><div><div></div><div></div><div></div></div></div>	Copy	<div><div><div></div><div></div><div></div></div></div>	Delete	56	5	2023-02-10 05:43:19	-1502570542
<div><div><div></div><div></div><div></div></div></div>	<div><div><div></div><div></div><div></div></div></div>	Edit	<div><div><div></div><div></div><div></div></div></div>	Copy	<div><div><div></div><div></div><div></div></div></div>	Delete	9	5	2023-01-07 00:24:27	1380644205
<div><div><div></div><div></div><div></div></div></div>	<div><div><div></div><div></div><div></div></div></div>	Edit	<div><div><div></div><div></div><div></div></div></div>	Copy	<div><div><div></div><div></div><div></div></div></div>	Delete	49	5	2025-09-30 13:09:02	1581127738

And looking at the `scores` table, we can see that each `session_id` shows up five times, and each `level_id` is valid:

You may have wondered what the `f` parameter is in the `refine` method. We can use it to generate more realistic data by using the various [generator functions](#) that Drizzle provides.

Let's apply some of these functions to our seeding script. We'll go over them below.

```
// /scripts/reseed.ts

...
await seed(db, schema).refine((f) => ({
  users: {
    count: 12,
    columns: {
      createdAt: f.date({ minDate: '2025-01-01', maxDate: '2025-02-01'
    }},
      dateOfBirth: f.date({ minDate: '1995-01-01', maxDate: '2005-12-
31' })
    },
    with: {
      userProfiles: 1,
      sessions: 5
    }
  },
  userProfiles: {
    columns: {
      firstName: f.firstName(),
      lastName: f.lastName({ isUnique: true }),
      title: f.valuesFromArray({
        values: ['Intern', 'Junior', 'Senior']
      })
    }
  },
  sessions: {
    columns: {
      // Between 30 minutes and 4 hours
      duration: f.number({ minValue: 60 * 30, maxValue: 60 * 60 * 4 })
    },
    with: {
      scores: 5
    }
  },
  levels: {
    count: 5
  },
  scores: {
    columns: {
      score: f.number({ minValue: 0, maxValue: 5000 }),

```



```

        // Between 1 minute and 12 hours
        timeTaken: f.number({ minValue: 60, maxValue: 60 * 12 }),
        // Between 0 and 1 with 2 decimal places
        accuracy: f.number({ minValue: 0, maxValue: 1, precision: 100 })
    }
}
}));
...

```

In this updated seeding script, we use the following generator functions:

- `f.date({ minDate, maxDate })`: Generates a random date between the specified minimum and maximum dates. We use this to generate a realistic date of birth for the users.
- `f.firstName()`: Generates a random first name for the user profile.
- `f.lastName({ isUnique: true })`: Generates a random last name for the user profile, ensuring that the last names are unique.
- `f.valuesFromArray({ values })`: Generates a random value from the specified array. We use this to generate a random title for the user profile.
- `f.number({ minValue, maxValue })`: Generates a random number between the specified minimum and maximum values. We use this to generate a realistic duration for the sessions, a score for the scores, and a time taken for the scores.
- `f.number({ minValue, maxValue, precision })`: Generates a random number between the specified minimum and maximum values, with the specified precision. We use this to generate a realistic accuracy for the scores.

You can find more generator functions in the [Drizzle documentation](#), should you need them.

Though the data is now more realistic, we still have a couple of ways to improve our data further. For example, we should ensure that the `created_at` column in the `score` table is set after the `created_at` column in the `sessions` table. Otherwise we'll be creating scores that didn't happen in a session, which doesn't make sense.

To make these changes, we should be generating data that depends on other data. `drizzle-seed` isn't too good at that. So we will instead need to do some manual seeding of the data, which we'll cover in the next section.

Manual seeding

As mentioned above, some of our data needs to be generated in a rigid way, and some of our data depends on other data. For example, we want to ensure that the `levels` difficulty gradually increases, and that the `created_at` column in the `scores` table is achieved within the timespan of the session it belongs to.

As `drizzle-seed` doesn't support this out of the box, we will need to manually insert some of the data. For some cases, we will have to query the dependent data first, and then use that data to generate the dependent data.

We'll be using [Faker.js](#) to help fill in the gaps where `drizzle-seed` isn't able to generate the data we need.

Faker.js

Faker.js is a popular library for generating fake data, and we can use it in conjunction with `drizzle-seed`.

To use Faker.js, we first need to [install it](#) as a dev dependency:

```
npm install @faker-js/faker --save-dev
```

Now we can import it in our seeding script and use it to generate the data we need.

To ensure that `faker` generates reproducible data, we can set a seed manually:

```
// /scripts/reseed.ts
















import { faker } from '@faker-js/faker';

// Function to reseed the database
async function reseed_db() {
  ...
  // Create a MySQL client and initialize Drizzle ORM
  const client = mysql.createPool(process.env.DATABASE_URL);
  const db = drizzle(client, { schema, mode: 'default' });
  faker.seed(1234);
  ...
}
```

Look at the [usage guide](#) and [API reference](#) to see what Faker.js can do for us.

Updating manually

Let's start off simple. We just want to manually [update](#) the `name` and `difficulty` of the created `levels` table, as they are rather nonsensical right now:

				id	name	difficulty
<input type="checkbox"/>		Edit		Copy		Delete
	1	Temur	-860113593			
<input type="checkbox"/>		Edit		Copy		Delete
	2	Rosaire	-713428047			
<input type="checkbox"/>		Edit		Copy		Delete
	3	Gayle	478161048			
<input type="checkbox"/>		Edit		Copy		Delete
	4	Cotton	1802265105			
<input type="checkbox"/>		Edit		Copy		Delete
	5	Diane	-116724874			

This gives us a good opportunity to write our first query that updates data in the database using Drizzle.

Below the `seed` function we add:

```
// /scripts/reseed.ts

async function reseed_db() {
  ...
  await seed(db, schema).refine(...);

  // We add this function call after the seeding process
  await updateLevels(db);
  ...
}
```

And then we implement the `updateLevels` function:

```
// Updates the levels table with sequential names and difficulties
async function updateLevels(db: MySQL2Database<typeof schema>) {
  // We define an array of levels with their names and difficulties
  const levels = [
    { name: 'Beginner', difficulty: 1 },
    { name: 'Intermediate', difficulty: 2 },
    { name: 'Advanced', difficulty: 3 },
    { name: 'Expert', difficulty: 4 },
    { name: 'Master', difficulty: 5 }
  ];





















  // For each level, update the name and difficulty
  for (let i = 0; i < levels.length; i++) {
    // We call the update method on the db object, passing in the
    schema.levels table
    await db.update(schema.levels)
      // We set the name and difficulty columns to the values from the
    levels array
```

```

        .set({
            name: levels[i].name,
            difficulty: i + 1
        })
        // We use the eq function to specify which row to update, based
on the id column
        .where(eq(schema.levels.id, i + 1));
    }
}

```

Executing the seed script, we have fixed the level table:

				id	name	difficulty
<input type="checkbox"/>		 Edit	 Copy	 Delete	1 Beginner	1
<input type="checkbox"/>		 Edit	 Copy	 Delete	2 Intermediate	2
<input type="checkbox"/>		 Edit	 Copy	 Delete	3 Advanced	3
<input type="checkbox"/>		 Edit	 Copy	 Delete	4 Expert	4
<input type="checkbox"/>		 Edit	 Copy	 Delete	5 Master	5

That looks much better!

Now let's move on to something a little more complex.

Selecting and updating using a Faker function

Similarly to how we updated the levels table, we can use a Faker.js function to generate random internet-like usernames for each user in the `users` table.

We first [select](#) the users from the `users` table, and then for each user, we will generate a random username using Faker.js and update the `username` column in the `users` table where the `id` matches the user's id.

```

// Sets random internet usernames for each user
async function updateUsernames(db: MySQL2Database<typeof schema>) {
    const users = await db.select().from(schema.users);
    for (const user of users) {
        await db
            .update(schema.users)
            .set({
                username: faker.internet.username().slice(0, 20)
            })
            .where(eq(schema.users.id, user.id));
    }
}

```

```
}  
}
```

Generating data that depends on other data

One of the issues with our current seeding script is that the `sessions` table might end up with sessions that were created before the user account was created, due to the random nature of the seeding process.

So, let's correct this by manually updating the `created_at` column in the `sessions` table to ensure that they're always sequentially created after the user account was created.

For this, we first have to [select](#) the users from the `users` table, and then for each user, we will select their sessions and update the `created_at` column based on some random, sequential offset from the user's `created_at` date:

```
// Adjusts the created_at column in the sessions table to ensure that  
sessions are created after the user account was created  
async function adjustSessionCreatedAt(db: MySQL2Database<typeof schema>) {  
  // Query the created users  
  const createdUsers = await db.select().from(schema.users);  
  
  // For each user...  
  for (const user of createdUsers) {  
    // Find their sessions  
    const userSessions = await db.query.sessions.findMany({  
      where: (sessions, { eq }) => eq(sessions.userId, user.id)  
    });  
  
    // Offset the sessions for that user  
    let currentCreatedAt = user.createdAt;  
    for (const session of userSessions) {  
      // Offset the createdAt by a random amount of time (up to 48  
hours)  
      const offset = faker.number.int({ min: 0, max: 60 * 60 * 48 *  
1000 });  
      const newCreatedAt = new Date(currentCreatedAt.getTime() +  
offset);  
      // Update the session's createdAt  
      await db  
        .update(schema.sessions)  
        .set({ createdAt: newCreatedAt })  
        .where(eq(schema.sessions.id, session.id));  
      currentCreatedAt = newCreatedAt;  
    }  
  }  
}
```

```

    }
  }
}

```

Cool. Let's use another faker function to ensure that each score is created within the timespan of the session it belongs to.

```

// Adjusts the created_at column in the scores table to ensure that scores
// are created in the timespan of the session
async function adjustScoresCreatedAt(db: MySQL2Database<typeof schema>) {
  // Query all sessions
  const sessions = await db.select().from(schema.sessions);

  // For each session...
  for (const session of sessions) {
    // Calculate the session's start and end times
    const sessionCreatedAt = session.createdAt;
    const sessionEndedAt = new Date(sessionCreatedAt.getTime() +
session.duration * 1000);

    // Find all scores for that session
    const scores = await db.query.scores.findMany({
      where: eq(schema.scores.sessionId, session.id)
    });

    // For each score, set a random createdAt time within the session's
    // timespan
    for (const score of scores) {
      const randomTime = faker.date.between({ from: sessionCreatedAt,
to: sessionEndedAt });
      await db
        .update(schema.scores)
        .set({ createdAt: randomTime })
        .where(eq(schema.scores.id, score.id));
    }
  }
}

```

Those are all the refinements we need to do for now.

Again, remember to not go too overboard with overdoing these refinements, and focus on normalizing the data that is most important for your application.

We mostly just wanted to show you how to refine the seeded data, and can always come back and refine the data later on.

Querying the database

Now that we have some realistic, usable data to show in our application, we want to know how to query the database to retrieve this data.

We've already written some basic queries in our seeding script, but now is a good time to read through the [Drizzle documentation](#). (In our web portal, for the most part we aren't altering any data, just reading it, so we'll mostly be using `select` queries.)

Most notably, we should know that there's two ways to query the database using Drizzle:

- **Query Builder:** This is the most common way to query the database, where we use the `db.select()` method to build a query step by step. It mimics SQL syntax closely.
- **Drizzle Queries:** This is what the `relations` we defined in our schema are used for. It allows us to query related data in a more intuitive way, using the `db.query` method. This is especially useful when we want to query data that has relations, such as users with their profiles or sessions.

We can use either method to achieve the same result. Depending on the situation, one method might be more convenient than the other.

In the next chapter, we'll be looking at implementing these queries in our SvelteKit application, so it's a good idea to familiarize yourself with the different query methods that Drizzle provides.

Wrapping up

In this chapter, we've started understanding ORMs conceptually to building a fully functional, relationally-structured database for our SvelteKit application.

We started by grasping why ORMs like Drizzle are valuable alternatives to raw SQL, then moved through the practical setup of connecting Drizzle to our MySQL database. From there, we built our database schema incrementally - beginning with a simple user table, then expanding to create meaningful relationships between users, profiles, sessions, scores, and levels. We learned to generate realistic test data efficiently, refining it to meet our specific needs, and finally explored how to query this data using Drizzle's flexible querying approaches.

What we've accomplished:

- Established a solid understanding of ORM benefits and Drizzle's TypeScript-first approach
- Created a production-ready database configuration with proper environment variable management
- Designed a relational database that accurately models a serious game's data requirements
- Built an automated seeding system that generates consistent, realistic data for development and testing

- Learned to query our database, readying us for data retrieval in our SvelteKit application

With our database foundation in place, we're now ready to bring this data to life in our SvelteKit application. In the next chapter, we'll learn how to safely integrate these database queries into our server-side routes, create dynamic pages that display user data, and build interactive components that visualizes raw information.

Visit [Chapter 4](#) to continue.