group-project # sprint

Sprint 2 — Prisma & Persistence

School of Engineering and Technology, University of Washington Tacoma
TCSS 460 — Client/Server Programming, Spring 2026

Due Date

Sunday, April 26, 2026, 11:59 PM

Sprint Narrative

Sprint 1 taught your API to speak. Sprint 2 teaches it to remember. Until now your API has been stateless — every request fetches fresh data from TMDB, nothing persists between calls. That's fine for search and browse, but the client is clear that the real product is **user-generated content**: ratings and reviews from real people, aggregated and queryable.

This sprint introduces your PostgreSQL database and Prisma, your new ORM. You'll design the schema for the data your team *does* own — users, ratings, reviews — and build full CRUD endpoints for ratings and reviews so the frontend team has something real to call. Remember the client's constraint: your database stores user-generated content only. **No movie or TV metadata** — TMDB remains the source of truth for that.

You'll also wire up the same authentication plumbing production APIs use: `Authorization` header, Bearer tokens, JWT verify middleware, `req.user` on every protected handler. Your API doesn't know how to talk to a real identity provider yet, so you'll drop in a **dev-only mint endpoint** — `POST /auth/dev-login` — that hands out tokens for local testing. Next sprint replaces that endpoint with the real Auth-Squared integration. Your middleware, routes, and handlers won't need to change.

MVP

By the end of this sprint, your API persists user-generated content in a PostgreSQL database through Prisma. Ratings and reviews support full CRUD through endpoints your team designed; writes are gated by a JWT from the dev-only mint endpoint. Your schema lives in source control as a Prisma migration, your OpenAPI docs cover the new endpoints, and your test suite verifies them.

Course Learning Objectives

This sprint contributes to the following course learning objectives:

- **LO 1:** Design and implement RESTful web APIs using a server-side framework
- **LO 2:** Model and access relational data using an ORM backed by a relational database
- **LO 6:** Transfer object-oriented programming skills to a new language and ecosystem
- **LO 7:** Collaborate in teams using version control workflows, sprint milestones, and code review

It also supports these course outcomes:

- **Inquiry and Critical Thinking** — designing a normalized schema, choosing relationships and constraints, and translating client requirements into a data model

---

🔧 Project Setup

Add the Dev Auth Module

This sprint introduces JWT-based authentication plumbing. The `dev-login` endpoint is a temporary stand-in for the real Auth-Squared integration coming in Sprint 3 — but the middleware you install this sprint stays. When real auth arrives, you'll swap one endpoint; your routes and middleware won't change.

One team member should run through these steps, then open a PR so the rest of the team gets the changes.

1. Install dependencies

```
npm install jsonwebtoken
npm install -D @types/jsonwebtoken
```

2. Download the module files

```
curl --create-dirs -o src/middleware/requireAuth.ts \
  https://raw.githubusercontent.com/UWT-SET-TCSS460-LECTURE-
MATERIALS/TCSS460-group-project-backend/main/modules/sprint-2-
dev-auth/requireAuth.ts

curl --create-dirs -o src/routes/devAuth.ts \
  https://raw.githubusercontent.com/UWT-SET-TCSS460-LECTURE-
MATERIALS/TCSS460-group-project-backend/main/modules/sprint-2-
dev-auth/devAuth.ts
```

3. Create a shared Prisma client

Create `src/lib/prisma.ts`:

```
import { PrismaClient } from '@prisma/client';
export const prisma = new PrismaClient();
```

4. Add a JWT secret to your `.env`

```
JWT_SECRET=<any long random string for local dev>
```

Never commit `.env`. Every team member generates their own secret locally.

5. Mount the dev-auth router in your server

In `src/app.ts` (where you register routes):

```
import devAuthRouter from './routes/devAuth';
app.use('/auth', devAuthRouter);
```

6. Seed an admin account

The `dev-login` endpoint find-or-creates regular users on demand, but admin accounts need to exist in advance. Write a Prisma seed script (`prisma/seed.ts`) that creates at least one admin user. Run it with `npx prisma db seed` after your initial migration.

What you're getting:

File	What It Does
`requireAuth.ts`	Middleware that reads the `Authorization: Bearer <token>` header, verifies the JWT with your `JWT_SECRET`, and attaches `req.user`
`devAuth.ts`	Router exposing `POST /auth/dev-login` — accepts a `username`, find-or-creates a user, signs a JWT with `{ sub, email, role }`

This is temporary

`dev-login` does not validate a password. Anyone hitting your API can claim any identity. This is a local-development stand-in for a real identity provider, and it will be removed in Sprint 3. Do not deploy this endpoint to a public URL.

Local Development Only

Sprint 2 runs entirely on your laptops — your database lives in Docker and your API serves `localhost`. Do not push this sprint's work to your Render service. Your Sprint 1 proxy deploy should keep running untouched so it continues serving the TMDB routes you demoed last sprint; the new `/ratings` and `/reviews` endpoints stay local until Sprint 3, when you'll connect to a hosted database and a real identity provider.

This is also why Sprint 2 submits to a feature branch instead of `main` — see "How to Submit" below.

User Stories

As a team, we want to design our data model before writing code so that we have a shared schema to build against.

Before you start coding, sit down as a team and design your Prisma schema. What tables do you need? What fields? What relationships? What constraints — uniqueness, required vs. optional, indexes? **Hard constraint from the client:** your database stores *user-generated content only* — users, ratings, reviews. **No movies or TV shows tables.** TMDB remains the source of truth for that metadata; your ratings and reviews reference TMDB identifiers, they don't duplicate TMDB data.

Design questions you own:

- What fields does a rating need beyond a score and a TMDB reference?
- Can a user rate the same movie twice, or is it an upsert?
- Do reviews have a title field, or just a body?
- How do you distinguish a rating on a movie from a rating on a TV show — separate tables, a discriminator column, or a composite key?
- What indexes make your GET endpoints fast?

Document your decisions. Your Prisma schema file is a good place for this, and it doubles as your migration source.

Guide

Prisma ORM — Schema definitions, migrations, relations, and queries
PostgreSQL Setup — Local PostgreSQL installation and connection

---

As a user, I want to submit a rating for a movie or show so that my opinion contributes to the community's picture of the content.

Given a valid JWT from `dev-login`, a user submits a score for a specific TMDB identifier. The handler reads `req.user` (never trusts a `userId` in the body) and attaches the rating to the authenticated user. Your team decides whether re-rating the same content creates a new row, updates the existing one, or returns an error.

As a user, I want to update and delete my own ratings and reviews so that I can correct mistakes or change my mind.

Update and delete endpoints verify that the authenticated user owns the record before modifying it. A user who tries to modify someone else's rating gets back a clear error (401 or 403 — your team decides the semantics and picks one consistently). Your team also decides how PUT behaves — full replacement, partial update, or PATCH instead.

As a user, I want to write a review for a movie or show so that I can share more than just a score.

A review carries more weight than a rating — a title (optional or required?), a body, maybe a timestamp. Your team designs the shape. Writes are gated by `requireAuth`; the author is always `req.user`.

As an admin, I want to delete inappropriate reviews so that the platform stays usable.

An admin can delete **any** review regardless of author. Your middleware distinguishes this from the regular delete path by inspecting `req.user.role`. Your team decides whether admin actions are soft-delete (hide but retain) or hard-delete (remove entirely) — and whether to expose this as a distinct admin route or a role check inside the existing DELETE handler.

As a visitor, I want to see ratings and reviews for a movie or show so that I can decide whether to watch it.

Public GET endpoints — no auth required. A visitor requests ratings or reviews for a given TMDB identifier and receives back a list or summary your team designed. Do you return raw records, aggregates (average score, review count), or both? Do you paginate? Do you sort by date, by score, or let the caller choose? Your team decides.

> **Guide**
>
> Pagination — Offset vs. cursor pagination, trade-offs and implementation

---

As a frontend developer, I want OpenAPI documentation and automated tests for every new endpoint so that I can integrate without reading source code and trust the API behaves as documented.

Every new route — ratings CRUD, reviews CRUD, dev-login, public GETs — appears in your `/api-docs` with accurate request and response schemas. Your Jest/Supertest suite covers success paths, error cases (unauthorized, not found, validation failures), and role-gated behavior (admin actions from a non-admin account). Tests should not call `dev-login` on

every request — extract a test helper that mints tokens directly with your `JWT_SECRET` and reuses them.

Guide

API Testing — Writing automated tests with Jest and Supertest
OpenAPI Documentation — Maintaining your OpenAPI spec with Scalar

---

Deliverables

- Prisma schema defines users, ratings, and reviews — no movies or TV tables
- Initial migration is committed to the repository under `prisma/migrations/`
- Seed script creates at least one admin account
- Ratings support create, read (single + list), update, delete
- Reviews support create, read (single + list), update, delete
- Public GET endpoints return ratings/reviews for a given TMDB identifier
- Writes and admin actions are gated by `requireAuth` and appropriate role checks
- `/auth/dev-login` is mounted and mints working JWTs
- OpenAPI documentation covers all new endpoints with request/response shapes
- Automated tests cover success and error cases for every new endpoint
- All team members have committed to the repository
- Meeting minutes document updated with sprint planning and any ceremonies

How to Submit

Sprint 2 ships on a feature branch, **not** `main`. Create a branch named `sprint-2-persistence` off `main`, do your work there (team members open PRs into `sprint-2-persistence`, not into `main`), and leave it unmerged by the due date. Your instructor grades from `sprint-2-persistence` on your GitHub Classroom repository; demos are run locally on your machine.

This is a one-sprint deviation from the usual "merge to `main` to submit" convention. Your Render deploy auto-ships whatever lands on `main`, and Sprint 2's database-backed routes can't run there yet — so we stage the work on a branch until Sprint 3 brings the cloud database and real auth online. At that point Sprint 3 folds `sprint-2-persistence` forward into `main` along with the cloud wiring.

Guide Reference

Guide	What It Covers
Prisma ORM	Schema-as-code, migrations, relations, type-safe queries
PostgreSQL Setup	Local PostgreSQL installation and connection strings
SQL Fundamentals	Background on what Prisma is abstracting
Pagination	Offset vs. cursor approaches to paging through list endpoints
API Testing	Jest + Supertest patterns for endpoint tests
Error Handling & Validation	Returning meaningful error responses
Routing and Middleware	Mounting routers and chaining middleware
OpenAPI Documentation	Documenting endpoints with Scalar

Supporting Documents

- Client Conversation — Original requirements discussion between the client and the technical lead
- Group Project Overview — Sprint schedule, architecture summary, and key principles
- Sprint 1 — TMDB proxy routes your CRUD endpoints coexist with

---

Gen AI & Learning: AI in Group Projects

AI coding assistants are permitted and encouraged. Every team member must be able to explain any code in your repository. During sprint reviews, you may be asked to walk through specific sections. "The AI wrote it" is not an explanation — understanding is the requirement.

This sprint is a particularly good place to use an agent: Prisma schema design, migration workflow, CRUD handlers, and test patterns are all well-understood territory that agents scaffold cleanly. Your job is to review, understand, and own the code — not to accept it blindly.

This assignment is part of TCSS 460 — Client/Server Programming, School of Engineering and Technology, University of Washington Tacoma.