group-project sprint

Sprint 3 — Auth & Hosted Database

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

Due Date

Sunday, May 3, 2026, 11:59 PM

Sprint Narrative

Sprint 2 taught your API to persist data, but two shortcuts kept it from looking anything like production: your database lived on your laptop, and anyone hitting your API could claim any identity through `dev-login`. Sprint 3 retires both. You'll integrate with the course's shared Auth² service — real RS256-signed tokens verified against its JWKS, scoped to your group's own audience in the ring — and you'll deploy the whole thing to a cloud platform with a hosted PostgreSQL database and your Prisma migrations applied in production. By the end of the week, your API has a public HTTPS URL, a real identity story, the start of the bug-report workflow, the first combined route that joins TMDB metadata with your community's ratings, and a downstream partner who's about to start depending on it.

About Auth²

Auth² is the course's instructor-hosted identity provider — not a proprietary protocol, just a standard **OAuth2 + OIDC** implementation. Everything you learn integrating with it — RS256 token verification, JWKS key rotation, audience-scoped access tokens, issuer validation — applies directly to any OAuth2 provider you'll meet in industry: Auth0, Okta, Keycloak, Google Identity Platform, Microsoft Entra. The name is ours; the patterns are industry-standard.

MVP

By the end of this sprint, your API is deployed to a public HTTPS URL backed by a hosted PostgreSQL database, with Prisma migrations applied in production. `dev-login` is gone; every protected route verifies real RS256 tokens from Auth² against its JWKS, scoped to your group's own audience. A frontend developer on another origin can mint a token from the Auth² admin portal, call your deployed API with it, and get back data — including your first combined route that returns a movie or show alongside what your community thinks of it.

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 3:** Implement authentication and authorization using token-based and federated identity patterns (e.g., JWT, OAuth2)
- **LO 5:** Deploy full-stack applications to cloud infrastructure
- **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 your role-gating strategy, evaluating cloud platforms and managed Postgres options, and choosing a deliberate audience name your whole team will live with
- **Communication/Self-Expression** — your OpenAPI spec and README become the contract a downstream partner group will build against next sprint

---

🔧 Project Setup

Wire your API to Auth²

Sprint 2's `dev-login` is being retired. Real tokens come from Auth² starting now. One team member should run through these steps, then open a PR so the rest of the team gets the changes.

1. Look up your group's audience

Your instructor has already registered an audience for each group in the `tcss460-sp26` tenant. Your audience name is `group-N-api`, where `N` is your group number (e.g., Group 3 → `group-3-api`). You'll wire this name into your middleware config, your CORS allowlist, your README, and your future FE wiring — it stays the same for the rest of the quarter.

2. Install the JWKS verification dependencies

```
npm install express-jwt jwks-rsa
```

3. Remove the dev-login module

Delete `src/routes/devAuth.ts`, the `app.use('/auth', devAuthRouter)` mount in `src/app.ts`, the `JWT_SECRET` env var (from `.env`, `.env.example`, README, and any docs), and `jsonwebtoken` from `package.json` if nothing else in your codebase imports it. Grep your repo for `JWT_SECRET`, `devAuth`, `dev-login`, and `jsonwebtoken` — nothing should remain.

4. Update environment variables

Add to your `.env` (and document in `.env.example`):

```
AUTH_ISSUER=https://tcss-460-iam.onrender.com
API_AUDIENCE=group-N-api
CORS_ALLOWED_ORIGINS=http://localhost:3000
```

Never commit `.env`. Document the shape in `.env.example` so new teammates can reproduce it.

5. Reference: `TCSS460-backend-3`

The Sprint 3 lecture demo shows the new `requireAuth` middleware shape — `expressjwt` + `jwks-rsa` plus a thin wrapper that copies `request.auth` onto `request.user` so your existing handlers don't change. Audience there is `backend-3-messages`; yours is `group-N-api` (from step 1). **Do not copy backend-3's audience name verbatim.**

Sprint 2 fallout

Removing `dev-login` breaks Sprint 2's test helper (which minted HS256 tokens with `JWT_SECRET`) and the seed script (which created admin accounts that no longer exist locally). Both are addressed in the user stories below — don't skip them.

User Stories

As a team, we want our API to verify real tokens issued by the shared Auth² service so that every authenticated user across the course's APIs is one user, identified the same way everywhere.

Replace the dev-login mint endpoint and `JWT_SECRET` with middleware that fetches Auth²'s public JWKS, verifies RS256 signatures, checks that the token's audience matches your group's audience name, validates the issuer, and rejects expired tokens. The `Authorization: Bearer <token>` header convention from Sprint 2 is unchanged — only the verification internals change.

Roles arrive on the token as PascalCase strings (`User`, `Moderator`, `Admin`, `SuperAdmin`, `Owner`). Your team decides how you gate routes:

- Exact match — "Admin only" rejects everyone except `Admin`
- Hierarchy — "Admin or above" accepts `Admin`, `SuperAdmin`, `Owner`
- Both — keep exact-match for narrow cases, use hierarchy for "any privileged user"

Pick consistently and document your choice. Look at `TCSS460-backend-3` for the middleware shape — your audience name is yours, not `backend-3-messages`.

Guide

Routing and Middleware — Mounting routers and chaining middleware

---

As a user, I want my actions to be tied to my Auth² account so that my ratings and reviews follow me across deployments and partner apps.

Auth²'s tokens carry a subject claim — a stable string identifier for the authenticated user. Your `User` table needs to track that claim as a unique field separate from your local primary

key. Add a `subjectId String @unique` column. Your existing `User.id` stays an integer; foreign keys from `Rating`, `Review`, and anything else you've built stay numeric.

The token itself carries only the subject, role, and (sometimes) email — not enough to render a useful profile. On the first authenticated request from any user, call `GET {AUTH_ISSUER}/v2/oauth/userinfo` with the same bearer token, and upsert a local row keyed by the subject claim with the enriched fields: `subjectId`, `username`, `email`, `firstName`, `lastName`. Store enough to answer "who is this?" without another Auth² round trip. **Call userinfo only on the upsert miss** — subsequent requests find the local row and skip the network call. Public routes (search, browse, public GETs) skip the upsert entirely — no DB write on a request that doesn't need one. See `TCSS460-backend-3`'s `resolveLocalUser` for the reference implementation, including sensible fallbacks when a field is missing from the userinfo response.

This sprint's migration **also** introduces the `Issue` model used by the next story. Ship both schema changes in a single Prisma migration so Sprint 4 doesn't need to touch the schema again — your downstream partner inherits a stable database. Your team designs the `Issue` model's fields and the `status` workflow; do that design work as part of this story even though the routes that exercise the table land in the next two stories.

Guide

Prisma ORM — Migrations, `@unique` constraints, and upserts

As a teammate, I want the test suite to keep working after we replace dev-login with real auth so that we can verify our routes without a real identity provider in the loop.

Sprint 2's test helper minted HS256 tokens with `JWT_SECRET`. That stops working the moment you switch to JWKS — the new middleware verifies against a public key fetched over the network, and your tests shouldn't depend on a real Auth² instance to mint against.

Replace the helper with a stubbed-middleware pattern: tests pass a header (your team picks the convention) carrying the desired claims; the stub attaches them as `request.user` and skips the real verification entirely. Real auth runs in production and Postman; tests stay fast and self-contained. The middleware itself is exercised end-to-end through your deployed API, not through unit tests.

Audit your existing Sprint 2 tests — anything that called the old helper has to migrate to the new pattern. Add tests for everything new this sprint: `POST /issues` validation, the enriched-detail combined route's happy path and 404, and at least one role-gated route now using a PascalCase string. Suite green before you deploy.

Guide

API Testing — Jest + Supertest patterns

As a visitor, I want to file a bug report against your API so that the team building on top of it can tell you when something is broken without finding you on Slack.

Your downstream partner group will start filing bugs against your API in Sprint 5. Build the public submission half of that workflow now: a route that accepts a bug report, persists it as an `Issue`, and returns confirmation. No authentication required — anyone with the URL can file. The admin-gated read and triage routes come in Sprint 4; the table needs to exist now so its migration is over and done with.

Your team designs the contract: what fields you require (title, description, repro steps, reporter contact?), what's optional, what your `status` enum looks like, and what shape your confirmation response takes. Validate input — reject anything malformed with a clear error and a useful status code. Don't accept a bug report with no description and no title.

Guide

Error Handling & Validation — Validating request bodies and returning meaningful errors

---

As a visitor, I want to see a movie or show's details alongside what your community thinks of it so that I can decide whether it's worth watching.

This is the API's flagship feature — the reason TMDB lives behind a proxy and your database stores ratings and reviews. A single request returns a movie or show's TMDB metadata together with your community's aggregate rating, recent reviews, and review count. One round trip, one response, one screen on the FE.

How you structure that combined response is your team's design call:

- Does the aggregate (average score, review count) live alongside the metadata in the same object, or in a sibling field?
- Do you include up to N recent reviews inline, or just summary stats with a separate route for the full list?
- Is the route public, authenticated, or both (public list + authenticated "did I rate this?" extension)?

- What happens when a movie or show has zero ratings yet — empty aggregate, null fields, or a different shape entirely?

But the route exists, it works, and it returns enriched data. Your FE team next quarter — or your downstream partner this quarter — should not have to make two calls and stitch the results together.

> **Guide**
>
> Proxy API — Calling third-party APIs from your routes
> Prisma ORM — Aggregations and joins

As a frontend developer, I want to call your API at a public HTTPS URL backed by a real database so that I can integrate against it from anywhere, not just from a teammate's laptop.

Pick a cloud platform — Render, Railway, Fly.io, or equivalent — and pair it with a managed PostgreSQL provider. Wire your app's `DATABASE_URL` to the hosted DB. Run `prisma migrate deploy` against production so this sprint's combined migration (subjectId + Issue) lands in the real database, not just the one in Docker on your laptop. Set the issuer URL and your audience name as environment variables in your hosting platform — never commit them.

Configure CORS. Your future Bug Tracker FE (Sprint 5) and your downstream partner's consumer app will call your deployed API from browsers running on different origins. Your allowlist should accept those origins, and your preflight responses should permit the `Authorization` header. Read the allowlist from an env var so you can change it without redeploying code.

> **Getting a test token for your deployed API**
>
> Use the **TCSS 460 Token Playground**: pick your group's audience, sign in (register the first time through), copy the access token. Paste it into `Authorization: Bearer <token>` when calling your deployed API.
>
> Access tokens expire in one hour. Click **Refresh** in the playground to mint a new one without re-signing in; after a tab reload, sign in again.

Test the deployed URL the same way you test locally: send the token in an `Authorization: Bearer <token>` header with `curl` or Postman, and confirm authenticated requests return the same data they return in dev. If local works and deployed doesn't, the bug is environmental: env vars, CORS, network, or migration state. **Walk through that diagnosis as**

a team before you ask for help — it's the same diagnosis you'll do every time something deploys for the rest of your career.

Guide

Deploying a Web API with a Database — Cloud platforms, hosted Postgres, environment configuration
Cloud Computing & Deployment (Week 4 reading) — The concepts behind what your hosting platform is doing for you

---

As a frontend developer, I want clear documentation, predictable errors, and a way to verify your API is alive so that I can build against it without reading your source.

A few small things make a deployed API feel like a real service instead of a half-finished prototype:

- **Health endpoint** — a public route that returns a small success payload. Hosting platforms use it for health checks; on-call humans use it to confirm the service is up. Your team picks the path (`/health`, `/healthz`, `/status` — pick one and document it).
- **Safe error handling** — your API never returns a stack trace or an internal exception message. Errors carry an HTTP status code and a structured response body the caller can act on. Internal error details get logged server-side with enough context to debug, but they don't leave your server. A `500` to a caller is one line; the same `500` in your logs has the full stack.
- **OpenAPI matches reality** — `/auth/dev-login` is gone from your spec. Your new routes (`POST /issues`, the enriched-detail route, your `/health` endpoint) appear with accurate request and response shapes. Your security scheme description names Auth² as the issuer and your audience name explicitly. Anything a partner needs to know to call your API correctly is in the spec.

Your downstream partner inherits this spec in Sprint 4 and builds against it in Sprint 5. The closer your spec matches your real behavior, the fewer bug reports they'll file.

Guide

Error Handling & Validation — Production-safe error responses
OpenAPI Documentation — Maintaining your spec with Scalar

Deliverables

- Audience name (`group-N-api`) documented in README and `.env.example`
- JWKS-based RS256 verification middleware in place; `dev-login` route, `JWT_SECRET`, and `jsonwebtoken` dep removed
- `subjectId String @unique` on `User`; `Issue` model created — both in a single Prisma migration
- `resolveLocalUser` (or your team's equivalent) used in handlers that need a local user row
- Test suite passes against the stubbed-middleware pattern; new tests cover `POST /issues`, the enriched detail route, and at least one PascalCase role gate
- `POST /issues` accepts and persists bug reports; input validated; response shape documented
- Enriched movie/show detail route returns TMDB metadata combined with your community's aggregate and recent reviews
- API deployed to a public HTTPS URL backed by hosted PostgreSQL
- `prisma migrate deploy` ran against production; production schema reflects this sprint's migration
- CORS allowlist configured via env var; preflight allows `Authorization` header
- Health endpoint returns a success payload at a documented path
- Error responses never leak stack traces or internal exception messages
- `/api-docs` reflects current state — no `dev-login` entry, new routes documented with examples, security scheme names Auth² as the issuer
- All team members have committed to the repository
- Meeting minutes document updated with sprint planning and any ceremonies

How to Submit

All work must be merged into `main` by the due date. Your instructor grades from the `main` branch of your GitHub Classroom repository.

Guide Reference

Guide	What It Covers
Routing and Middleware	Mounting routers, chaining middleware, ordering matters
Prisma ORM	Migrations, `@unique` constraints, upserts, aggregations
Proxy API	Calling third-party APIs from your routes
Pagination	Offset vs. cursor — relevant if your enriched route lists reviews
API Testing	Jest + Supertest with stubbed-middleware patterns
Error Handling & Validation	Production-safe error responses; request body validation
OpenAPI Documentation	Maintaining your spec with Scalar
Deploying a Web API with a Database	Cloud platforms, hosted Postgres, env configuration

Supporting Documents

- Client Conversation — Original requirements discussion; the "magic part" quote is the basis for the flagship combined route this sprint
- Group Project Overview — Sprint schedule, ring topology, partner handoff timing
- Sprint 2 — Schema, CRUD, and dev-login plumbing this sprint replaces
- Authentication & Authorization Concepts (Week 5 reading) — JWT, JWKS, OAuth2, RBAC fundamentals
- Cloud Computing & Deployment (Week 4 reading) — What your hosting platform and managed Postgres provider are doing for you

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 the kind of work agents handle well: middleware swaps, Prisma migrations, deploy configuration, and OpenAPI updates are all well-trodden ground. **The risks are configuration drift and silent acceptance** — an agent will happily generate middleware that verifies a different audience than your team registered, or env-var names that don't match your hosting platform's UI, or CORS rules that look right but don't preflight. Read what the agent produces. Test the deployed result, not just the local one. The deploy diagnosis ("it works locally but not deployed — why?") is the most valuable skill this sprint teaches; don't let an agent take it from you.

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