group-project # sprint

Sprint 4 — Partner-Ready Completion

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

Due Date

Sunday, May 10, 2026, 11:59 PM

Sprint Narrative

Last week your API got real: hosted Postgres, RS256 auth verified against Auth²'s JWKS, the first combined route joining TMDB with your community's reviews. This week is about finishing the product — and getting ready to be its first consumer. Next sprint, your team becomes the first front-end on your own API, building the Bug Tracker FE on top of the `/issues` rows you've been collecting since last Sunday. That means this week ships the admin half of the bug-tracker workflow (list, read, triage, delete) and the two remaining combined routes that complete the "magic part" the client described. You'll also expose the small authorship surface your future front-end will need — self-list routes so a logged-in user can see their own reviews and ratings, and an `author` identity inline on every review and rating response so a partner can render a review feed without a separate `/users/:id` round trip. Your downstream partner doesn't start consuming your API until Sprint 6, but the contract they'll build against — your OpenAPI spec, your README, your CORS allowlist — hardens this week. No new migrations; the schema's been done since last Sunday. By Sunday, the API is product-complete and partner-ready.

MVP

By the end of this sprint, an admin can list, read, triage, and delete bug reports filed against your API; both remaining combined routes return the enriched user-rated-items and community-ranked data the client described; an authenticated user can list their own reviews and ratings, and every review or rating in any response carries its author inline; your OpenAPI spec at `/api-docs` covers every route in the API with realistic example payloads and error shapes; and your partner-facing README tells a downstream developer everything they need to start calling your deployed API. End of week, the API is partner-ready: your team picks it up next sprint as the Bug Tracker FE's backend, and your downstream partner picks it up in Sprint 6 for their consumer app.

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
- **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 combined-route shapes that match the client's "magic part," choosing a community-ranking query strategy that won't fall over at scale, and committing to PATCH-vs-PUT semantics your partner can rely on
- **Communication/Self-Expression** — your OpenAPI spec and partner-facing README stop being internal working notes this week and become the contract a downstream developer reads to integrate

---

User Stories

As an admin, I want to see every bug report filed against our API so that I can spot patterns, prioritize fixes, and answer "did anyone already report this?" without going through Slack.

Sprint 3 stood up the public submission half (`POST /issues`). This sprint ships the read half — admin-gated routes that return the queue and a single report's detail.

Your team designs the contract. Do admins get pagination by default, or do you return everything? Filtering by status — and which statuses can be combined? Sort order — newest first, oldest first, status-grouped? What does "admin" mean for these routes — exact `Admin`, or any role at or above `Admin`? Pick consistently with the strategy you committed to in Sprint 3 and document it.

The route gates on the same JWKS-verified token from last sprint; no new auth plumbing.

Guide

Pagination & Filtering — Designing list endpoints

As an admin, I want to update a bug report's status and delete reports we've resolved or that are spam so that the queue stays meaningful.

Triage means status changes — your team designed the `status` enum back in Sprint 3 (`open`, `in-progress`, `resolved`, `wontfix`, whatever you picked). PATCH walks a report through that workflow. DELETE removes it.

Decide what fields PATCH accepts. Decide what DELETE returns. Decide whether `PATCH /issues/:id` accepts a partial body or requires the full report shape — if you read PATCH as "merge this partial change," that's idiomatic, but write it down somewhere your partner will see, because PUT-vs-PATCH semantics aren't universally agreed on.

Validate input. A PATCH body that doesn't change anything is fine; a PATCH body with an unknown status string is a 400 with a clear error.

Guide

Error Handling & Validation — Validating partial updates and returning meaningful errors

---

As a user, I want to see every movie and show I've rated alongside its TMDB metadata so that I can browse my own taste history.

This is the second of three required combined routes — the flagship enriched detail in Sprint 3 was the first. Same pattern: one request, one response, joined data. The user's ratings rows live in your database; the metadata lives behind TMDB. Your route fans out, joins the results, and returns a single shaped response.

Your team designs the contract:

- One TMDB call per rated item, or batch them (TMDB supports batched lookups for some endpoints)?
- What's the response shape — a flat list of enriched ratings, or grouped (movies vs. shows)?
- Pagination? Sort options (by rating, by date rated, by title)?
- What about items that have been removed from TMDB — skip, mark as missing, or fail the whole request?

Authenticated route. The `sub` claim from the token tells you whose ratings to fetch — the URL doesn't carry a user id. Don't trust a `userId` query parameter even if a caller sends one.

> **Guide**
>
> Proxy API — Calling TMDB from your routes; handling errors when the upstream is down
> Prisma ORM — Joins and aggregations

As a visitor, I want to see what your community is rating highest right now so that I can find something worth watching without knowing what to search for.

The third required combined route. Your team picks one of two angles, or both with separate routes:

- **Top-rated** — items with the highest community average rating, with a minimum review count to keep one-rave-review outliers out
- **Most-reviewed** — items with the most ratings/reviews regardless of score

Either way, the route joins your aggregate (average score + review count) with TMDB metadata for each item. Public route — visitors can see the discovery feed without an account.

The aggregate is a database query, not an in-memory computation. `SELECT AVG(score), COUNT(*) ... GROUP BY movie_id HAVING COUNT(*) >= N` belongs in your DB, not in TypeScript. Your team also decides: do you query live each request, cache the result for some window, or materialize a view? Whichever you pick, document it in the README so a partner knows what response time to expect.

> **Guide**
>
> Prisma ORM — Aggregations, GROUP BY, and HAVING

As a user, I want to see my own reviews and ratings, and as a visitor I want every review and rating to show who wrote it, so that the review feed reads like a community and not an anonymous dump.

The JWT already carries the user's stable subject claim — that's enough. Your team doesn't need a `/users` endpoint surface; identity belongs to Auth², and `resolveLocalUser` already syncs the local row from Auth²'s userinfo on first sign-in. What's missing is exposing what you already have through the review and rating surface. Two pieces:

- **Self-list routes.** An authenticated user can list their own reviews and ratings without passing a user id in the URL. The handler reads `request.user.sub`, looks up the local row, returns that user's records. Don't trust a `userId` query parameter even if a caller sends one. Your team picks the route shape (`GET /reviews/me`, `GET /reviews?author=me`, `GET /me/reviews` — pick the convention that fits the rest of your API) and documents it.
- **Author identity on review and rating responses.** Every review or rating row your API returns includes an `author` object with at least the user's stable id and a display name. Public review feeds become readable ("by @alice") without a separate lookup; your downstream partner doesn't need a `/users/:id` round trip to render. Your team picks the field name (`author`, `user`, `reviewer`) and what fields go inside.

Display name comes from the local user row your `resolveLocalUser` populated in Sprint 3 — typically `username` or a constructed `firstName lastName`. If a row is missing a display name (because a user signed in but the userinfo enrichment failed), document the fallback your API uses.

As a frontend developer (your downstream partner), I want a complete, accurate OpenAPI spec so that I can build my consumer app against your contract without reading your source.

By the end of this sprint, every route in your API appears in the spec at `/api-docs`. Every request body has a schema. Every response has at least one realistic example payload. Every error response — 400, 401, 403, 404, 500 — has a documented shape. Your security scheme names Auth² as the issuer and your specific audience.

This means walking the whole API, not just this sprint's new routes. Sprint 1's TMDB proxy routes, Sprint 2's ratings/reviews CRUD, Sprint 3's auth-gated routes and flagship combined route, this sprint's admin routes and two combined routes — all of it. Anything currently undocumented or out-of-date gets fixed.

The spec stops being your team's working document this week. From this sprint onward, your team adopts a "spec changes ship in the same PR as code changes" rule — drift between the

spec and reality is what costs your partner an afternoon, every time it happens.

Guide

OpenAPI Documentation — Schema modeling, examples, and error responses

---

As a frontend developer (your downstream partner), I want a partner-facing README and a CORS configuration that lets my app call your API so that I can integrate against your deployed API on day one.

Your README needs to answer the questions a downstream developer will actually have:

- Where is the API deployed? (your public HTTPS URL)
- How do I get a token? (link to the Token Playground; your group's audience name explicitly)
- What endpoints exist? (link to `/api-docs`)
- What origins are CORS-allowed, and how do I get my origin added?
- Where do I file bug reports? (link to your Bug Tracker FE — it ships next sprint; document the planned URL now so the README is correct on Monday)
- What known limits or quirks should I expect? (rate limits, TMDB cache behavior, anything you discovered the hard way)

Your CORS allowlist needs to accept your downstream partner's consumer-app origin (Sprint 6+ deployments). If your partner hasn't picked a hosting platform yet, accept their localhost dev origin and document the production add as a one-line env-var change.

Partner-ready bar: by end of week, a downstream developer should be able to read your README, hit your deployed API with a token from the Playground, and make their first successful authenticated request without asking you anything. The actual cross-group swap happens at the start of Week 7 — your team carries the API into Sprint 5 first as its own Bug Tracker FE backend; your downstream partner picks it up at the start of Sprint 6 for their consumer app.

Guide

API Testing — Verifying CORS preflight against your deployed API

Deliverables

- [ ] Admin-gated `GET /issues` (list) and `GET /issues/:id` (detail) live; pagination/filtering/sort behavior documented
- [ ] Admin-gated `PATCH /issues/:id` walks reports through your `status` workflow; partial-vs-full body semantics documented
- [ ] Admin-gated `DELETE /issues/:id` removes a report; response shape documented
- [ ] Combined route — user's rated items joined with TMDB metadata — live and authenticated against the `sub` claim
- [ ] Combined route — community-ranked list (top-rated and/or most-reviewed) — live, public, with a SQL-side aggregate
- [ ] Authenticated self-list routes (your team's chosen shape) return the caller's own reviews and ratings; no client-supplied user id is trusted
- [ ] Every review and rating response — public feeds, self-list, and the user's-rated-items combined route — includes an `author` object with stable id and display name
- [ ] OpenAPI spec at `/api-docs` covers every route in the API: schemas, realistic example payloads, documented error shapes for 400/401/403/404/500
- [ ] Security scheme in the spec names Auth² as the issuer and your audience name explicitly
- [ ] Partner-facing README answers all six questions from the handoff story
- [ ] CORS allowlist accepts your downstream partner's consumer-app origin (or their dev origin, with a documented env-var update path for production)
- [ ] Test suite green; new tests cover the admin `/issues` routes, the two new combined routes, and at least one PATCH validation case
- [ ] 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
Pagination & Filtering	Designing list endpoints — defaults, sort, filter
Error Handling & Validation	Validating partial updates; production-safe error responses
Proxy API	Calling TMDB from your routes; upstream error handling
Prisma ORM	Aggregations, GROUP BY, HAVING, joins
API Testing	Jest + Supertest patterns; CORS preflight verification
OpenAPI Documentation	Schema modeling, examples, error responses
JWT Verification with Auth²	Reference for the role helpers gating your admin routes

Supporting Documents

- Sprint 3 — Auth, hosted database, schema finalization, the `Issue` model and `POST /issues` your admin routes complete this sprint
- Group Project Overview — Sprint schedule, ring topology, partner handoff timing
- Client Conversation — The "magic part" quote is the basis for the two combined routes this sprint
- Authentication & Authorization Concepts (Week 5 reading) — JWT, JWKS, OAuth2, RBAC fundamentals
- TCSS 460 Token Playground — Mint tokens for your audience to test admin routes and partner integration

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's risks are different from Sprint 3's. The plumbing is mostly in place; the work is shape-design and contract-hardening, both of which agents are confident about and frequently wrong about. An agent will produce a perfectly reasonable PATCH handler that silently accepts unknown status values. It will write OpenAPI examples that look right but don't match what your route actually returns. It will generate a README section that's plausible but wrong about your audience name or your CORS allowlist. **Read what the agent produces against your real running API, not against the agent's claim of what your API does.** The OpenAPI freeze in particular is a place where small drift compounds — your partner pays for every undetected mismatch in Sprint 6.

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