group-project sprint

Sprint 7 — Consumer App: Rate & Review

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

Due Date

Sunday, May 31, 2026, 11:59 PM

Sprint Narrative

Last week was reads — your consumer app could sign a visitor in, search the partner's API, browse what's popular, and render a detail page. This week the app actually does something for the user who signed in. The rate button works. The review form posts. Users see their own ratings and reviews on a profile page, edit a rating they want to change, delete a review they regret. Every write goes out with the bearer token from the NextAuth session and respects the partner's contract — your app does not own the routes; the partner does. This is the sprint where your consumer app stops being a browser for someone else's data and starts being a place where your users have a presence.

MVP

By the end of this sprint, a signed-in user on your consumer app can submit a rating and a review against your upstream partner's API, see those items rendered back on the partner's enriched detail page (community aggregate + their own contribution), and view their own ratings and reviews on a profile page. They can edit or delete content they own. Every authenticated request attaches the access token from the NextAuth session. Signed-out visitors see inert affordances — a "Sign in to rate" placeholder where the rate button lives for signed-in users — not a broken control.

Course Learning Objectives

This sprint contributes to the following course learning objectives:

- **LO 3:** Implement authentication and authorization using token-based and federated identity patterns (authenticated writes with a bearer token from the NextAuth session)
- **LO 4:** Build interactive front-end applications using a component-based framework (Next.js — forms, controlled inputs, session-aware UI)
- **LO 7:** Collaborate in teams using version control workflows, sprint milestones, and code review (cross-group bug filing through your partner's tracker)

It also supports these course outcomes:

- **Inquiry and Critical Thinking** — reading the partner's spec for write routes you didn't design, debugging the bearer-token attachment when a write returns 401, and deciding what "successful submit" should *look* like in the UI (optimistic update vs refetch, inline vs modal edit)
- **Communication/Self-Expression** — building forms that communicate state clearly (submitting, success, validation error, network error), and filing structured bug reports through your partner's Bug Tracker FE when their write routes don't match their spec

---

User Stories

As a user, I want to rate a movie or show so that I can record how I felt about it.

From the detail page (or wherever your team puts the rate control), a signed-in user submits a rating against your upstream partner's rating route. Your fetch attaches the bearer token from the NextAuth session. After a successful submit, the detail page reflects the rating — either optimistically updated client-side, or refetched from the partner's enriched detail route. Whether the control is a 1–10 slider, a 5-star widget, half-stars, or buttons is your team's design call; the value range is bounded by what the partner's contract accepts.

If the rating you submit comes back shaped differently than the partner's spec promised, or you get a 500 on what looks like a valid request, that's a bug for *them* — file it through their Bug Tracker FE (see Story 6).

Guide

Consuming a Web API — Writes specifically: headers, request bodies, status codes, error response shapes

---

As a user, I want to write a review so that I can share my thoughts with other people who watched it.

A review form on the detail page (or its own route). Read your partner's OpenAPI spec to learn what their review payload requires — usually title and body, sometimes just body, occasionally a target field that names the movie or show. Wire form validation, a visible submit state, and an error path for the case where the partner returns a 400 or 401.

After a successful POST, the user's review appears in the partner's review list for that title. Accessible form labels and focus management are part of the bar this sprint — this is the most form-heavy thing your app does.

Guide

Accessibility — Form labels, error association, focus management on submit/error
Styling & Component Libraries — Form patterns from the FE-2 lecture demo (MUI `TextField`, `Button`, validation states)

As a user, I want to edit or delete content I own so that I can change my mind or fix a mistake.

Ratings and reviews you submitted should expose edit and delete affordances when *you're* the author. PUT/DELETE against your partner's routes with the bearer token attached. Whether edit happens inline, in a modal, or on a dedicated route is your team's design call.

If the partner's contract does not actually expose update or delete on a resource you thought it would, file a bug against them; do not work around it by hacking the API.

As a user, I want a profile page that shows everything I've rated and reviewed so that I can see my activity in one place.

Hit your partner's "own content" routes — typically something shaped like `/users/me/ratings` and `/users/me/reviews`, possibly an enriched combined route that joins TMDB metadata onto the user's items. Render the list(s) on a `/profile` route (or whatever path you chose for the sign-in surface in Sprint 6).

This is also the natural surface for the edit/delete affordances from Story 3 — a user looking at their own list of reviews is the moment they're most likely to want to fix one. The page shape is bounded by what your partner returns; if their "my ratings" route doesn't enrich with TMDB metadata, you either fetch the metadata yourself per row or accept a thinner UI.

As a visitor, I want write affordances to be inert when I'm not signed in so that the app doesn't appear broken when I haven't logged in.

Sprint 6 said "no write affordances rendered yet." Now they exist — but only for signed-in users. A signed-out visitor on a detail page sees a "Sign in to rate" placeholder where the rate control lives for signed-in users, and a similar treatment on the review form (or it's simply absent with a sign-in CTA). Do not render a button that 401s when clicked.

The `useSession` hook (or the equivalent server-component check) is the signal. Signed-out copy still needs to be readable, keyboard-reachable, and clearly labeled as a sign-in prompt — not a disabled mystery control.

Guide

Accessibility — Disabled vs. unavailable controls, labeling sign-in prompts so screen readers explain what's happening

---

As a team, we want our consumer app to feel like one product so that a user moving between views isn't jarred by inconsistent choices.

By Sprint 7 you have real surface area — search, browse, detail, profile, sign-in, the rate control, the review form, edit/delete affordances. That's enough surfaces for inconsistency to creep in: text buttons on one view and icon buttons doing the same thing on another, three different card spacings, two different shades of "primary" depending on which file an agent generated, body copy in three font sizes nobody chose on purpose.

Walk the app end-to-end as a team and make deliberate, consistent choices — button styles, spacing, typography scale, color usage, empty states, loading states, where errors appear. Pick a small set of components and reuse them. If you're using MUI, lean on the theme; if you're hand-rolling, factor out a few shared components and import them everywhere instead of restyling at every call site.

This is **not** the final polish — that's Sprint 8. The bar this week is *intentional*, not *perfect*. Inconsistencies that exist because nobody noticed yet are the ones to fix; choices the team made deliberately are fine even if they're rough.

Guide

Styling & Component Libraries — Theme tokens, shared components, the MUI patterns from FE-2 that let you change one value and have it propagate

---

As a frontend developer, I want to file structured bugs in my partner's Bug Tracker FE when their write routes misbehave so that I get a real channel for contract friction.

Writes are where API contracts break first — 500s on what looks like valid input, 403s when a token decodes cleanly, response shapes the spec didn't promise, missing fields after a PUT, ratings that come back as strings when the spec said number. The course's standing policy from Sprint 6 still applies: don't message your partner on Slack.

File the bug in their Bug Tracker FE — the URL you collected in Sprint 6 — with the endpoint you hit, repro steps, what you expected, and what you got. One bug per issue; don't pile three unrelated problems into one report.

Deliverables

- Signed-in users can submit a rating against the partner's API; the rating renders back on the detail page
- Signed-in users can submit a review against the partner's API; the review appears in the partner's review list for that title
- Signed-in users can edit and delete ratings and reviews they own
- Profile page renders the signed-in user's own ratings and reviews (one route, both lists, or two routes — your call)
- All authenticated requests attach the bearer token from the NextAuth session in the `Authorization` header
- Signed-out visitors see inert affordances (sign-in prompts, not 401-bound buttons) anywhere a write would otherwise appear
- Form validation and error states present on the review form (and the rating control if it has a failure mode)
- App has been walked end-to-end as a team for design and UX cohesion; button styles, spacing, typography, and component choices are consistent across views (not final polish — that's Sprint 8)

- At least one structured bug filed in your partner's Bug Tracker FE *if* their write routes misbehaved (none required if their API worked cleanly)
- All team members have committed to the consumer-app repository this sprint
- 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. The consumer app continues in the **same Sprint 6 GitHub Classroom group repository** — there is no new Sprint 7 repo. Sprints 6, 7, and 8 all ship from the same consumer-app repo.

Your BE Stays Live

Your group's deployed back-end continues running through the rest of the quarter — your downstream partner is consuming it for *their* consumer app this sprint, and writes from their side mean more bugs are likely to land in your `/issues` queue this week than landed last week. Triage them via Postman or Prisma Studio against your admin-gated routes. Their development is gated on your responsiveness the same way yours is gated on your upstream partner's.

Guide Reference

Guide	What It Covers
Consuming a Web API	Fetch lifecycle for writes (POST/PUT/DELETE), headers, request bodies, status codes, error response shapes, reading a partner's spec for write contracts
Authentication with NextAuth	Reading the access token off the session, attaching it to authenticated fetches, `useSession` for signed-out gating, server-component session reads
Next.js	App Router, server vs client components, forms, environment variables
React Fundamentals	Controlled inputs, form state, conditional rendering for signed-in vs signed-out UI

Guide	What It Covers
Accessibility	Form labels, error association, focus management, disabled vs. unavailable controls
Styling & Component Libraries	MUI form patterns (`TextField`, `Button`, validation states) from the FE-2 lecture demo

Supporting Documents

- Sprint 6 — The sign-in + read foundation this sprint builds writes on top of
- Group Project Overview — Sprint schedule, ring topology, partner handoff timing
- Client Conversation — The product Buddy described in Week 1; rate-and-review is where the consumer app starts looking like what the client asked for
- TCSS460-frontend-2 lecture demo — Next.js 15 + Auth.js v5 + MUI v7 reference app; the authenticated-fetch pattern in this demo is the pattern your writes use
- TCSS 460 Token Playground — Mint a token for your partner's audience out-of-band to verify their write routes accept it before debugging through NextAuth

Gen AI & Learning: AI in Group Projects

AI coding assistants are permitted and encouraged. The "every team member must be able to explain any code in your repository" bar is the standing norm and applies in full this sprint. During sprint reviews, expect to be asked to walk through your authenticated-fetch pattern, your form submission flow, your `useSession` gating, and how you handle write errors. "The agent wrote it" is not an explanation.

Writes are also where the agent is most likely to produce code that *looks* correct but isn't — bearer tokens missing from a fetch, optimistic updates that don't reconcile on error, form validators that pass on empty strings. Reading what the agent produced for your write paths carefully, and testing the failure cases by hand, is the right amount of effort.

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