group-project # sprint

Sprint 5 — Bug Tracker FE

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

Due Date

Sunday, May 17, 2026, 11:59 PM

Sprint Narrative

This week the groups swap and your team becomes a front-end team for the first time. Before you take on consuming a partner's API, you build a small front-end on the API you know best: your own. The job is a public bug-report form that posts to your own `POST /issues`. No login, no triage dashboard, no admin views. Your downstream partner — paired with your API at the start of this week — will use this form during Sprints 6–8 every time something breaks. Triage stays on your side of the wall: you'll work the queue from Postman or Prisma Studio against the admin routes you shipped in Sprint 4.

This sprint is also where AI scaffolding becomes the explicit workflow. You haven't been taught React or Next.js yet — that arrives this week and across the rest of the quarter. What you're practicing now is the *prompt-and-iterate* loop: feed an agent a real spec, read what it produces, decide what to keep. Code-level understanding follows in the Week 7+ readings, lectures, and Check-Off 6. For *this* sprint, the bar is "I can describe my workflow," not "I can explain every line."

MVP

By the end of this sprint, every team member has individually scaffolded a Bug Tracker FE with an AI agent against your team's OpenAPI spec, the team has met to compare those builds, and the team has shipped **one** final version — either the strongest individual build, or a composite the team (or an agent) merged from the best features. The final build lives in your team's Sprint 5 GitHub Classroom group repository, deployed to a public URL, posting successfully to your live `POST /issues`, with realistic success / validation-error / network-failure states. Each member's individual workflow writeup — prompts, agent output, what was kept and cut — is committed to the final repo. NextAuth and OAuth2 do not appear this sprint; they arrive next week with the consumer app.

Course Learning Objectives

This sprint contributes to the following course learning objectives:

- **LO 4:** Build interactive front-end applications using a component-based framework (Next.js)
- **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 the prompt that gets a useful first cut from an agent, judging which of multiple AI-scaffolded builds is the strongest, and deciding what to merge versus throw out
- **Communication/Self-Expression** — your individual workflow writeup (the prompts, the iterations, what you kept and cut) is the artifact this sprint produces; the team's final README is what your downstream partner reads

---

How This Sprint Works

This sprint is structured differently from earlier sprints. You haven't learned React or Next.js yet — that comes in the readings and lectures across Week 7 and beyond. The point of Sprint 5 is the **AI-driven scaffolding workflow** itself.

The flow is:

1. **Individually.** Each team member scaffolds their own Bug Tracker FE with an agentic coding tool (Claude Code, Cursor, etc.) against the team's OpenAPI spec. Solo work —

local-only on your machine, or in any personal repo of your choosing. There is no GitHub Classroom repo for individual scaffolds; bring your build (and your workflow writeup) to the team meeting.

2. **Together.** The team meets, walks through each member's build, and either:
   - **Picks** the strongest individual build to be the team's final, or
   - **Merges** the best features from multiple builds into a composite. An agent can drive this merge — that's a fair use of AI here.
3. **Ship once.** The team's selected or merged build goes into the team's Sprint 5 GitHub Classroom repository (one repo per group, not per member). That repo — and only that repo — is what's deployed and what your downstream partner uses.
4. **Document.** Each member writes up their individual AI workflow (prompts used, what came back, what was kept and cut, what they'd do differently next time). Those writeups are committed to the final team repo so the prompt-and-iterate process is captured as the learning artifact.

> **GitHub Classroom — Sprint 5 Group Repo**
>
> The Sprint 5 GitHub Classroom assignment is a **group** assignment, like every other sprint — one repository per team, not one per student. The starter repo is **empty** — your team drops the final selected-or-merged build in fresh.
>
> **Accept the assignment:** Sprint 5 — Bug Tracker FE on GitHub Classroom
>
> One team member accepts first and creates the team; remaining members accept the same link and join the existing team.

Working with the Agent: Context Over Prompting

The way to get the most out of an agent on this sprint isn't a longer prompt — it's a richer working directory. The skill being practiced is **preparing the context** so the agent can do most of the work without you spelling everything out.

Try this:

1. **Start in a fresh, empty directory.** Don't open the agent on top of an existing repo full of unrelated context — that just gives it stuff to be confused by.
2. **Drop the materials in as files.** Save this sprint document as `sprint-5.md` in the directory. Save your team's OpenAPI spec as `openapi.yaml` (or `openapi.json`) next to it. Add anything else useful — your deployed BE URL in a `README.md`, an example `POST /issues` request body, **a sketch of the form you have in mind**.
3. **Open your agent of choice** — Claude Code, OpenAI Codex, VS Code's agent mode, Cursor, whatever it's called this month — and point it at the directory.
4. **Prompt short.** Something like *"Read `sprint-5.md` and `openapi.yaml` in this folder. Build the Bug Tracker FE described in the sprint doc against the API in the spec."* That's it. No 400-word prompt, no architecture lecture in the chat.
5. **See how little you can prompt.** Resist the urge to over-explain. If the agent goes off-track, the right fix is usually adding a file to the context or correcting a specific output — not writing a longer system message.

Prompting tells the agent *what to do*; context tells it *what it's working on*. Most of the value is in the context.

User Stories

As a visitor (your downstream partner), I want to file a bug report against your API without creating an account so that reporting friction is zero.

The form is the entire app for an unauth visitor. They land on the URL, fill out the fields, submit, and see a confirmation. Field set should match the request body your `POST /issues` already accepts — required vs. optional follows the OpenAPI spec.

The route this app calls is the same public `POST /issues` you stood up in Sprint 3. No JWT, no auth headers — your BE accepts unauthenticated reports by design.

Guide

Consuming a Web API — Fetch, request shapes, response handling

As a visitor, I want clear feedback when my submission succeeds, fails validation, or fails because the API is down so that I know whether my report actually got through.

A real user hits all three states. Success: confirmation message, form clears or redirects. Validation failure: inline errors next to the offending fields, surfaced from the response your API returned. Network or server failure: a non-cryptic message that doesn't suggest the user did something wrong, and that preserves what they typed so they can retry.

Guide

Accessibility — Form labels, error association, focus management on submit

---

As a teammate, I want to scaffold my own Bug Tracker FE solo with an agentic coding tool against our team's OpenAPI spec so that I get hands-on time with AI-driven scaffolding before the underlying tech is taught.

This is your individual work for the sprint. Open Claude Code, Cursor, or your team's tool of choice. Feed it your team's `/api-docs` spec (or paste the relevant slice). Ask for a Next.js app with a single public form that posts to `POST /issues`. Iterate — prompt, read what comes back, prompt again to fix or extend. Take the build as far as you reasonably can on your own.

You are **not** expected to fully understand the code the agent produced this week. React and Next.js are introduced in the Week 7+ readings, lectures, and Check-Off 6. The skill being practiced now is the workflow itself: writing a prompt that gets a useful first cut, recognizing when to redirect the agent, and judging when "good enough" is good enough.

Where your build lives is your call — local-only on your laptop, or in any personal repo of your choosing. There's no GitHub Classroom repo for individual scaffolds. Either way, bring the build (and your workflow writeup) to the team meeting.

Guide

Using a Coding Agent — Briefing the agent, reading its output, knowing when to redirect
AI Tools Setup — Getting an agentic tool installed and connected to your repo

---

As a team, we want to compare each member's individual scaffold and either pick the strongest one or merge the best features into a composite so that the final build reflects the team's collective best judgment.

The team meeting is the heart of this sprint. Each member walks through what they built, what their prompt sequence looked like, and what they think worked or didn't. Two valid paths from there:

- **Pick.** One member's build is clearly the strongest — adopt it as the team's final.
- **Merge.** Several builds have different strengths (one's form layout is cleanest, another's error handling is more thoughtful, a third's project structure is tidier). Combine them — by hand or by handing all the builds to an agent and prompting for the composite. Either is fair.

Capture the decision in your meeting minutes: which builds were considered, what the strengths were, and why you picked or merged the way you did.

As a teammate, I want to write up my individual AI workflow — what I prompted, what the agent produced, what I kept and cut — so that the team's final repo captures the prompt-and-iterate process as a learning artifact.

Your writeup is short and concrete. What did your first prompt look like? What did the agent produce on the first cut? What was good, what missed? What was your second prompt — a redirect, an extension, a fix? What did you ultimately keep, what did you throw out, and what would you prompt differently next time?

This is what gets graded for understanding this sprint, not the code. "I prompted X, got Y, kept Z because…" is the explanation form.

Each member's writeup goes into the final team repo (in the README, or in a `WORKFLOWS.md` linked from the README — your team's call). Your downstream partner doesn't need to read these — but you and your future self do.

As a frontend developer, I want the form to talk to our deployed API (not localhost) so that the deployed FE actually works end-to-end.

Environment variables. Your team picks the variable name (`NEXT_PUBLIC_API_URL` is the Next.js convention for a value the browser needs to see) and points it at your deployed BE URL in production and your local dev URL in development. No hardcoded URLs in component code. The Deploying Next.js guide walks the env-var setup for both Vercel and Render — the same patterns work whichever host your BE landed on.

CORS lives on the BE side. Your `POST /issues` route's CORS allowlist needs to accept the deployed FE's origin — that's a config change on your BE repo, not the FE one. Add the FE origin, redeploy the BE, verify the preflight in browser DevTools.

Guide

Deploying Next.js — Environment variables on Vercel and Render

---

As a team, we want the final Bug Tracker FE deployed to a public URL so that our downstream partner can actually use it during Sprints 6–8.

Vercel or Render — your team's choice. Push to `main` of the team FE repo, deploy auto-runs, public URL lives. Only the final selected/merged build is deployed; individual scaffolds stay in their personal repos or local.

Test against the deployed FE talking to the deployed BE — not localhost-to-localhost. That's the configuration your partner will hit, so that's the configuration that needs to work.

Guide

Deploying Next.js — Vercel + Render walkthrough, env vars, build commands

---

As a frontend developer (your downstream partner), I want to know where to find your Bug Tracker FE without asking so that bug reports don't go through Slack.

The URL goes in your partner-facing README (the one you wrote in Sprint 4) and gets sent to your downstream partner directly. Your partner's consumer app — built in Sprints 6–8 — will link to this URL from a "Report a Bug" button or similar.

Don't make your partner search. The first time they break something against your API, they should know exactly where to file it.

Deliverables

Individual (each team member):

- Solo AI scaffold of a Bug Tracker FE — local-only or in any personal repo of your choosing (no GHC repo for individual scaffolds)
- Workflow writeup — prompts you used, agent output, what you kept and cut, what you'd change next time

Team:

- Team meeting minutes capturing each member's build, the comparison, and the pick-or-merge decision
- One final build selected or merged from individual scaffolds (agent-driven merge is acceptable)
- Sprint 5 GitHub Classroom group repository accepted (separate from your BE repo) — final build lives here
- Final build deployed to a public URL (Vercel or Render)
- Final build's form posts successfully to your live `POST /issues` — verifiable from your BE side via Postman or Prisma Studio
- Success / validation-error / network-failure states each have visible UI feedback
- Final build talks to your deployed API via environment variables — no hardcoded URLs in component code
- BE CORS allowlist updated to accept your deployed FE origin; preflight verified end-to-end
- All individual workflow writeups committed into the final team repo (in README or `WORKFLOWS.md`)
- Final FE URL added to your partner-facing README (in the BE repo) and sent to your downstream partner
- All team members have committed to the final team FE 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. The Bug Tracker FE has its **own Sprint 5 GitHub Classroom group repository**, separate from your BE repo — link it from your group's BE repo README so it's discoverable. Individual scaffolds (local or in personal repos) are not graded for code quality; they're graded together with your workflow writeup.

Triage Stays Server-Side This Sprint

Sprint 5 is intentionally scoped to the public report form only — there's no admin or triage UI. To work the queue, your team uses the admin-gated `/issues` routes you shipped in Sprint 4 from Postman, or browses the `Issue` table in Prisma Studio. NextAuth and OAuth2 arrive next sprint with the consumer app.

There will not be a future sprint that builds a triage UI on this Bug Tracker FE. If you want one, build it yourself. With remaining tokens this week — or, more realistically, in a fresh session once Week 7's NextAuth content lands — you can extend this FE into a full bug-triage tool. The shape of that work:

- **Add NextAuth (Auth.js)** and configure it against Auth² as an OAuth2 provider — the issuer is `https://tcss-460-iam.onrender.com`, the audience is your `group-N-api` name, the client ID/secret come from your group's pre-seeded consumer client in the `tcss460-sp26` tenant.
- **Give yourself admin on your own BE.** Auth² hands every student a `User`-role token — admin determination is entirely your BE's responsibility. Promote your own user in your local `User` table (Prisma Studio is the fastest way: open the row, flip the role column to whatever your team's admin gate checks). The token from Auth² will keep saying `User`; your BE's middleware decides who's an admin from the local row.
- **Sign in** through NextAuth and store the access token on the session; pass it as `Authorization: Bearer <token>` on every outbound call to your BE.
- **Call your admin-gated `/issues` routes** — `GET /issues` for the list, `GET /issues/:id` for detail, `PATCH /issues/:id` for status updates, `DELETE /issues/:id` to clear out spam or resolved reports. These are the routes you shipped in Sprint 4.
- **Hide your triage pages behind a FE route guard.** Since the JWT's `role` claim is always `User`, the FE has to ask your BE who the caller is — typically by calling a `/me` (or equivalent) endpoint that returns the local user row's role. Cache the result on the NextAuth session and gate the route on it. Remember: front-end route guards are a UX convenience, not a security boundary. Your BE's admin middleware is the actual trust boundary; the FE guard just keeps the routes from rendering for users who'll get a 403 anyway.
- **Build the triage UI** — list view (sort/filter by status), detail view (full report + repro steps), status dropdown wired to PATCH, delete confirmation wired to DELETE, and a sign-out flow.
- **Add a sign-in / sign-out surface** — even a single button on a `/dashboard` route is enough.

Feed the agent enough context that your prompts stay short. Drop your OpenAPI spec, this sprint document, the JWT Verification with Auth² guide, the Authentication & Authorization Concepts reading, and your group's audience + client ID into the working directory. Then ask the agent to extend the FE with NextAuth, an admin-gated triage view, and the routes above. Same workflow as the public form — context first, then a short prompt.

None of this is required for Sprint 5 to be complete, and none of it is graded. It's the natural next step if you want a real tool instead of a Postman habit.

Guide Reference

Guide	What It Covers
Next.js	App Router, routing, components, fetch, environment variables
React Fundamentals	Components, props, state, controlled inputs
Consuming a Web API	Fetch lifecycle, error modes, CORS preflight
Accessibility	Form labels, error association, focus management
Deploying Next.js	Vercel + Render walkthrough, env vars, build commands
Using a Coding Agent	Briefing the agent, reading its output, knowing when to redirect
AI Tools Setup	Getting an agentic tool installed and connected to your repo

Supporting Documents

- Sprint 4 — The admin `/issues` routes and partner-facing README that this sprint's FE plugs into
- Group Project Overview — Sprint schedule, ring topology, partner handoff timing
- Client Conversation — The bug-tracker workflow the client described
- Evolution of Web Programming (Week 6 reading) — Where Next.js fits in the SPA / SSR / framework arc

Gen AI & Learning: AI in Group Projects

AI coding assistants are permitted and encouraged — and this sprint is specifically about that workflow. You'll lean on an agent to scaffold a Next.js app from your OpenAPI spec before you've been formally taught Next.js. That's intentional: the goal this week is the *prompt-and-iterate* loop, not full code-level understanding.

The standard course-wide bar — "every team member must be able to explain any code in your repository" — is **paused for Sprint 5 only**. It reactivates in Sprint 6 as the Week 7+ readings, lectures, and Check-Off 6 land the underlying React/Next.js concepts. By Sprint 6, the expectation that you can walk through your code is back to normal.

What you DO own this sprint: your individual workflow writeup (what you prompted, what came back, what you kept and cut, what you'd do differently next time), the team's selection-or-merge decision and the rationale behind it, and a deployed FE that actually works end-to-end against your API. "The agent built it" is fine for the code; it is not fine for the workflow narrative.

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