assignment # auth # check-off # express # jwt

Check-Off 5 — Auth²

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

Description

In this check-off, you will work with a lecture demo repository (`backend-3`) that already integrates with our course OAuth2 provider, **Auth²**. You will mint a real access token, observe the four characteristic auth outcomes against existing protected routes (`200` , `401` for invalid token, `401` for wrong audience, `403` for insufficient role), trace the verification code, and then add a single **hello-world-style role-gated route** of your own with OpenAPI documentation and tests. The new route's logic is intentionally trivial — the focus is the auth flow, not the business logic.

Learning Objectives

By completing this check-off, you will:

- Mint a real RS256 access token from Auth² and inspect its claims (`sub` , `aud` , `iss` , `exp` , `role`)
- Observe and explain the four characteristic auth outcomes: valid `200` , invalid token `401` , wrong audience `401` , insufficient role `403`
- Trace `requireAuth` (verify → attach user → error handler) and the three role helpers in source
- Wire `requireAuth` and `requireRoleAtLeast` into a new Express route
- Document a protected route with OpenAPI's `bearerAuth` security scheme
- Write tests that exercise an authed route using the project's middleware-stub pat

Course Learning Objectives

This check-off contributes to the following course learning objectives:

- **LO 1:** Design and implement RESTful web APIs using a server-side framework
- **LO 3:** Implement authentication and authorization using token-based and federated identity patterns

It also supports these course outcomes:

- **Inquiry and Critical Thinking** — reasoning from a single observable response code (`401` vs `403`) back to the specific verification step that produced it

---

Before You Begin

Ensure you have completed:

- **Read** the Week 5 concept reading: Authentication & Authorization
- **Read** the JWT Verification with Auth² guide
- **Docker Desktop installed and running** — PostgreSQL runs in a container
- **Node.js 22+** installed
- **An API testing tool installed** — Postman, Thunder Client (VS Code extension), or similar

Guides for This Check-Off

These guides cover the concepts and patterns you will use:

- JWT Verification with Auth² — `requireAuth`, role helpers, `resolveLocalUser`, debugging 401/403
- Routing & Middleware — how middleware chains compose
- API Testing — Jest + Supertest patterns
- OpenAPI Documentation — documenting protected routes

Project Setup

Requirement 1: Clone and Run Backend-3

TCSS 460 Backend 3

github.com/UWT-SET-TCSS460-LECTURE-MATERIALS/TCSS460-backend-3

Express + TypeScript + PostgreSQL + Prisma + Auth² JWT verification. The Sprint 3 reference implementation. Verifies RS256 tokens against the public JWKS at `https://tcss-460-iam.onrender.com/.well-known/jwks.json` for audience `backend-3-messages`.

Clone the repository and install dependencies:

```
git clone https://github.com/UWT-SET-TCSS460-LECTURE-MATERIALS/TCSS460-backend-3.git
cd TCSS460-backend-3
npm install
```

Set up the environment file (the example file is pre-filled for the SP26 deployment):

```
cp .env.example .env
```

The defaults that matter for this check-off:

Variable	Default	What it does
`PORT`	`3001`	Server port
`AUTH_ISSUER`	`https://tcss-460-iam.onrender.com`	The Auth² issuer — used to fetch JWKS and validate the `iss` claim
`API_AUDIENCE`	`backend-3-messages`	The expected `aud` claim — tokens for any other audience are rejected

Start the database, run the migration + seed, and start the dev server:

```
docker compose up -d
npm run db:setup
npm run dev
```

Verify Everything Works

- `GET http://localhost:3001/health` — returns `{ "ok": true }`
- `GET http://localhost:3001/v2/messages` — returns a paginated list (this route is **public** — no token required)
- Open `http://localhost:3001/api-docs` in your browser — interactive API documentation

Requirement 2: Mint an Access Token

You will sign in to the **Token Playground** to mint a real RS256 access token from Auth².

TCSS 460 Token Playground

tcss460-token-playground.onrender.com

Steps:

1. Open the Token Playground in your browser.
2. From the audience dropdown, select `backend-3-messages`.
3. Click **Sign In** — a popup opens against Auth².
4. Sign in (or register if this is your first time — see the JWT guide §8 for first-time setup notes).
5. After the popup closes, the main window shows your access token, a decoded view of header + payload, and an expiry countdown.

Inspect the decoded claims and locate each of these:

- `sub` — your stable user ID inside Auth²
- `aud` — must be `backend-3-messages`
- `iss` — must be `https://tcss-460-iam.onrender.com`
- `exp` — expiry timestamp (1 hour from `iat`)
- `role` — your role on this tenant (a freshly-provisioned account is `User`)

Copy the access token. You will paste it into your API testing tool's `Authorization: Bearer <token>` header in the next requirement.

Token expires in 1 hour

If you get partway through the check-off and start seeing unexpected `401` responses, your token may have expired. Click **Refresh** in the Token Playground to mint a new one without re-entering credentials.

Requirements

Requirement 3: Observe the Four Auth Outcomes

Reference

See the JWT Verification with Auth² guide — especially the debugging decision tree in §9.

You will hit a protected route four times to observe the four characteristic auth outcomes. Use `POST /v2/messages` with a valid request body — that route is wrapped in `requireAuth`, so it is sensitive to every part of the token verification.

A valid body needs a non-empty `content` field — `subject` and `priority` are optional:

```
{ "content": "hello from check-off 5" }
```

Outcome A — Valid token → `201`

Send the request with `Authorization: Bearer <your-token>`. You should get `201 Created` with the new message in the response body.

Outcome B — Missing/invalid token → `401`

Send the same request with **no** `Authorization` header (or with `Bearer not-a-real-token`). You should get `401 { "error": "Invalid or missing token" }`. Explain to your checker which middleware in `requireAuth` produced this response.

Outcome C — Wrong audience → `401`

Go back to the Token Playground and mint a **second** token, this time selecting a different audience from the dropdown — any `group-N-api` will do. (You may need to log out and back in to swap the audience selection cleanly.) Send the original `POST /v2/messages` request with this new wrong-aud token in the `Authorization` header. You should get `401 { "error": "Invalid or missing token" }`. The token itself is cryptographically valid and was issued

by the same Auth² instance — it was just minted for a different audience than backend-3 expects, so `verifyJwt` rejects the `aud` claim.

Outcome D — Insufficient role → `403`

This requires a route protected by a role gate. You will build one in Requirement 5 — come back to Outcome D after that, OR demonstrate it now against your group's Sprint 3 backend if that already has a role-gated route. The contract: a `User`-role token hitting a route that requires `Admin` (or higher) gets `403 { "error": "Insufficient permissions" }` — **not** `401`. The token is valid; the role is not high enough.

Requirement 4: Trace the Verification Code

Open these files in `src/middleware/` and `src/auth/`. Be ready to point to them on screen during your check-off.

`src/middleware/requireAuth.ts` — locate:

- The exported `requireAuth` constant. Confirm it is an **array of three handlers**: `verifyJwt`, `attachUser`, `handleAuthError`. Why an array? (Hint: Express middleware chains.)
- `verifyJwt` — uses `expressjwt` + `jwks-rsa` to fetch the public key from the JWKS endpoint and verify the RS256 signature, audience, and issuer.
- `attachUser` — reads the verified claims from `request.auth` and copies them onto `request.user` as a typed `AuthenticatedUser`.
- `handleAuthError` — catches `UnauthorizedError` from `verifyJwt` and returns the `401 { "error": "Invalid or missing token" }` response.
- The three role helpers: `requireRole(role)` (exact match), `requireRoleAtLeast(minRole)` (minimum-rank gate), `hasRoleAtLeast(role, minRole)` (boolean for in-handler checks).
- The role hierarchy constant — confirm the order is `User < Moderator < Admin < SuperAdmin < Owner`.

`src/auth/resolveLocalUser.ts` — locate:

- That this is a **plain async helper**, not middleware. It returns `Promise<UserModel>`.
- The fast path (line ~19): `prisma.user.findUnique({ where: { subjectId } })`.
- The slow path (line ~57): on first call for a new `sub`, fetches `{ sub, email, name, role }` from `${AUTH_ISSUER}/v2/oauth/userinfo` and upserts a local `User` row. This is how every authed JWT gets a corresponding row in the BE-3 database keyed by `sub`.

Requirement 5: Add a Role-Gated Hello Route

Reference

See JWT Verification with Auth² §6 (role helpers) and §4 (route shapes) for the exact middleware-chain pattern.

Add a new route that requires authentication **and** at least the `Admin` role. The handler logic is intentionally trivial — return a small JSON object derived from the authed user. The point is the auth flow.

Route contract:

Method	Path	Auth	Role Required	Response
`GET`	`/v2/hello`	Yes	`Admin` or higher	`200 { "message": "Hello, <email>!", "role": "<role>" }`

Files to create:

- `src/controllers/v2/hello.ts` — a handler function that reads `request.user` (set by `attachUser`) and returns a JSON object shaped like:

```
{ message: `Hello, ${request.user.email}!`, role: request.user.role }
```

- `src/routes/v2/hello.ts` — a router with one `GET /` route. The middleware chain must include both `requireAuth` and `requireRoleAtLeast('Admin')` before the handler.

File to edit:

- `src/routes/v2/index.ts` — mount your new router under `/hello`. Follow the pattern used for `messages` and `users`.

Now demonstrate Outcome D from Requirement 3: call `GET /v2/hello` with your `User`-role token. You should get `403 { "error": "Insufficient permissions" }`. The verification step succeeded — `attachUser` ran and `request.user` was set — and then `requireRoleAtLeast('Admin')` rejected the request because `User` ranks below `Admin`.

How would you turn the 403 into a 200?

You do not need to do this for credit, but be ready to explain it: an Admin-role token would be required. In a real deployment, an instructor or tenant Admin would change your role on the Auth² admin portal, you would mint a fresh token, and the next call to `GET /v2/hello` would return `200`. Caching note: the role is a JWT claim, so a token minted **before** the role change will still see the old role until it is refreshed or expires.

Requirement 6: Document Your Route with OpenAPI

Reference

See the OpenAPI Documentation guide for YAML syntax. The relevant security scheme is already declared at the top of `openapi.yaml`:

```yaml
components:
  securitySchemes:
    bearerAuth:
      type: http
      scheme: bearer
      bearerFormat: JWT
```

Add a documentation entry for `GET /v2/hello` to `openapi.yaml`. Look at how an existing protected route (e.g., `POST /v2/messages`) is documented for the pattern. Your entry must include:

- A short `summary` and `description`
- A `security` block referencing `bearerAuth: []`
- The `200` response with the response body schema
- The `401` response (missing/invalid token) and `403` response (insufficient role)

Restart the dev server and confirm your route appears in the interactive docs at `http://localhost:3001/api-docs` with a lock icon indicating it is protected.

Requirement 7: Write Tests

Reference

See the API Testing guide. Open `tests/setup.ts` and `tests/helpers.ts` to see the project's middleware-stub pattern — `requireAuth` is mocked, and tests inject a synthetic `request.user` by setting an `x-test-user` header (built by the `authHeader({ sub, role })` helper).

Create `tests/v2/hello.test.ts`. Mirror the style of `tests/v2/messages.test.ts`. Your tests should cover:

- `GET /v2/hello` with no `x-test-user` header → `401`
- `GET /v2/hello` as a `User`-role caller → `403`
- `GET /v2/hello` as an `Admin`-role caller → `200` with `message` and `role` fields in the response body
- `GET /v2/hello` as an `Owner`-role caller → `200` (role hierarchy: Owner ranks above Admin)

Run the suite:

```
npm test
```

Your total test count must be **higher than 71** (the starting count).

Peer Check-Off

Find a classmate to verify your work.

For the Checker

The checker has 10 points to award. There is no rubric — use your judgment. Did they do the work? Can they explain it? Award points based on completeness and understanding — not perfection.

Check-Off Checklist

#	Verify	✓
1	Server starts with `npm run dev`, Docker database is running, and `GET /v2/messages` (public) returns a	☐

#	Verify paginated list	✓
2	Student can show a freshly-minted access token from the Token Playground and identify the `sub`, `aud`, `iss`, `exp`, and `role` claims in the decoded payload	☐
3	**Outcome A:** `POST /v2/messages` with a valid token and valid body returns `201`	☐
4	**Outcome B:** `POST /v2/messages` with no `Authorization` header returns `401 { "error": "Invalid or missing token" }`	☐
5	**Outcome C:** A token minted from the Token Playground for a different audience (e.g., `group-1-api`) returns `401` when used against backend-3	☐
6	**Outcome D:** A `User`-role token hitting `GET /v2/hello` returns `403 { "error": "Insufficient permissions" }` (not `401`)	☐
7	Student can point to `requireAuth` in `src/middleware/requireAuth.ts` and identify the three handlers (`verifyJwt`, `attachUser`, `handleAuthError`) and what each one does	☐
8	Student can explain the difference between `401` and `403` in their own words	☐
9	A new file `src/routes/v2/hello.ts` exists, mounted in `src/routes/v2/index.ts`, with a chain that includes `requireAuth` and `requireRoleAtLeast('Admin')`	☐
10	`GET /v2/hello` is documented in `openapi.yaml` with a `security: [bearerAuth: []]` block and shows a lock icon at `/api-docs`	☐
11	`npm test` passes and the total test count is higher than 71. The new test file covers no-token `401`, User-role `403`, and Admin-role `200`	☐

#	Verify	✓
12	Student can explain what `resolveLocalUser` does on the first authed request for a new `sub` versus subsequent requests	☐

How to Submit

1. Complete the check-off requirements above.
2. Find a classmate to be your checker.
3. Demo your work — walk them through each item on the checklist.
4. The checker has 10 points to award. There is no rubric — use your judgment. Did they do the work? Can they explain it? Award points based on completeness and understanding — not perfection.
5. Both of you fill in the shared spreadsheet linked from the Canvas assignment: the student enters their name in Column A, the checker enters their name in Column B, and the checker enters the score in Column C.

Canvas — Check-Off 5: Auth²

Guide Reference

Guide	What It Covers
JWT Verification with Auth²	`requireAuth`, role helpers, `resolveLocalUser`, debugging 401/403
Routing & Middleware	Express Router, file organization, middleware chains
API Testing	Jest + Supertest patterns, the middleware-stub mock
OpenAPI Documentation	Documenting protected routes with `bearerAuth`
Authentication & Authorization (concept reading)	The theory behind tokens, audiences, roles, and the four outcomes

Gen AI & Learning: Using AI for This Check-Off

You are welcome to use AI coding assistants to help you write your hello route, OpenAPI entry, and tests. However, your checker will ask you to explain why a `wrong-aud` token returns `401`, why a `User`-role token returns `403`, and what each of the three handlers in `requireAuth` does. If you cannot explain the auth flow, the check-off is not complete. Use AI as a tool, but make sure you understand every line.

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