assignment # check-off # database # express # prisma

Check-Off 3 — Prisma Intro

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

Due Date

Sunday, April 26, 2026, 11:59 PM

Description

In this check-off, you will work with a lecture demo repository that already has a working Prisma setup — two models, seed data, and full CRUD endpoints. You will add a **new model** to the Prisma schema, write seed data, create **four CRUD endpoints** for it, add Zod validation, document the routes, and write tests. The model is intentionally simple — the goal is to practice the full pipeline from schema to working API, not to design a complex data model.

Learning Objectives

By completing this check-off, you will:

- Define a Prisma model and run a migration to apply it to a running database
- Seed a table with initial data using a Prisma seed script
- Create a Zod validation schema and wire it into Express middleware
- Build four CRUD endpoints that use Prisma Client for database operations
- Handle Prisma errors (record not found) with appropriate HTTP status codes
- Document new endpoints in an OpenAPI specification
- Write tests for new endpoints following existing patterns

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 2:** Model and access relational data using an ORM backed by a relational database
- **LO 6:** Transfer object-oriented programming skills to a new language and ecosystem

It also supports these course outcomes:

- **Inquiry and Critical Thinking** — tracing the full pipeline from schema definition to working endpoint

---

Before You Begin

Ensure you have completed:

- **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:

- Prisma ORM — Schema definition, migrations, Prisma Client CRUD, error handling
- Error Handling & Validation — Zod validation, error responses, status codes
- Routing & Middleware — Express Router, file organization, middleware chains
- API Testing — Jest + Supertest patterns

Project Setup

Requirement 1: Clone and Set Up Backend-2

TCSS 460 Backend 2

github.com/UWT-SET-TCSS460-LECTURE-MATERIALS/TCSS460-backend-2

Express + TypeScript + PostgreSQL + Prisma demo. Contains v1 (raw SQL) and v2 (Prisma ORM) implementations of the same API for comparison.

Clone the repository and install dependencies:

```
git clone https://github.com/UWT-SET-TCSS460-LECTURE-MATERIALS/TCSS460-backend-2.git
cd TCSS460-backend-2
npm install
```

Set up the environment file:

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

Start the database and run the full setup:

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

This starts a PostgreSQL container, runs the Prisma migration, generates the Prisma Client, and seeds the database with 25 users and 500 messages.

Start the development server:

```
npm run dev
```

> **Verify Everything Works**
>
> Make these requests with your API tool:
>
> - `GET http://localhost:3000/v2/users` — should return a paginated list of users
> - `GET http://localhost:3000/v2/messages` — should return a paginated list of messages
> - Open `http://localhost:3000/api-docs` in your browser — interactive API documentation
>
> If any of these fail, check that Docker Desktop is running and the database container is healthy (`docker compose ps`).

Requirements

Requirement 2: Explore Existing Code

Before building anything new, explore the existing v2 implementation to understand the patterns you will follow. Open and review these files:

Data layer:

- `prisma/schema.prisma` — two models (`User`, `Message`) with a one-to-many relation
- `prisma/seed.ts` — how seed data is created using `createManyAndReturn` and `createMany`

Validation:

- `src/middleware/validation.ts` — Zod schemas (`MessageBodySchema`, `UserBodySchema`), the `validate` factory function, and exported middleware (`validateMessageBody`, `validateNumericId`, etc.)

There is a second file, `src/middleware/validation.legacy.ts`, that implements the same validation by hand without Zod. Comparing the two shows you why Zod is worth using.

Routes and controllers:

- `src/routes/v2/messages.ts` — how routes map HTTP methods to controller functions with validation middleware in the chain
- `src/controllers/v2/messages.ts` — how controller functions use Prisma Client (`findMany`, `create`, `update`, `delete`) and handle Prisma error codes (`P2025` for not found)
- `src/routes/v2/index.ts` — how route files are mounted onto the v2 router

The project also has `src/routes/v1/` and `src/controllers/v1/` directories with raw SQL versions of the same endpoints. You will work exclusively in v2 (Prisma) for this check-off, but glancing at a v1 controller shows you what Prisma replaces.

Tests:

- `tests/v2/messages.test.ts` — how tests mock Prisma and use Supertest to make requests

Run the existing test suite to confirm it passes:

```
npm test
```

You should see **57 tests passing** across 4 test suites.

Requirement 3: Define a New Prisma Model, Migrate, and Seed

> **Reference**
>
> See sections 2–4 of the Prisma ORM guide for schema syntax, migrations, and seeding.

Add a model to `prisma/schema.prisma`. The model should be simple — just an `id` and a single string field. For example, a `Saying` with a `content` field:

```
model Saying {
  id      Int    @id @default(autoincrement())
  content String
  @@map("sayings")
}
```

You are free to choose a different name and field — the only requirement is that it has an auto-incrementing integer `id` and at least one string field.

Run a migration to apply the change to your database:

```
npx prisma migrate dev --name add-sayings
```

This generates a SQL migration file in `prisma/migrations/` and regenerates the Prisma Client with types for your new model. Take a look at the generated SQL file — it shows you exactly what Prisma is doing under the hood.

Add seed data to `prisma/seed.ts`. Add at least 10 rows so your GET endpoint has data to return. Follow the existing seed pattern — clear the table first, then create rows:

```
await prisma.saying.deleteMany();
await prisma.saying.createMany({
  data: [
    { content: 'The early bird catches the worm' },
    { content: 'A journey of a thousand miles begins with a single step' },
    // ... at least 10 total
  ],
});
```

Re-run the seed:

```
npm run prisma:seed
```

> **Verify with Prisma Studio**
>
> Run `npm run prisma:studio` to open a visual browser for your database. You should see your new table with seed data in it.

Requirement 4: Add Zod Validation

Reference

See the Error Handling & Validation guide for Zod schema patterns and the validation middleware factory.

Before building your endpoints, you need validation middleware to protect them. Here are the four endpoints you will create in the next requirement:

Method	Path	Operation	Request Body
`GET`	`/v2/sayings`	List all (paginated)	None — uses `page` and `limit` query parameters
`POST`	`/v2/sayings`	Create one	`{ "content": "..." }`
`PUT`	`/v2/sayings/:id`	Update one	`{ "content": "..." }`
`DELETE`	`/v2/sayings/:id`	Delete one	None

POST and PUT both accept a JSON body with your model's field(s). Without validation, a client could send an empty body, a missing field, or the wrong type — and the error would surface as a cryptic database error instead of a clean `400` response. Zod schemas define the expected shape once, and the validation middleware rejects bad requests before they reach your controller.

Open `src/middleware/validation.ts` and add a Zod schema for the request body. Follow the existing pattern — look at how `UserBodySchema` and `validateUserBody` are defined:

1. **Define the schema** — validate that the string field is present and non-empty (e.g., `z.string().trim().min(1)`)
2. **Export a type** using `z.infer<>` — this gives you a TypeScript type derived from the schema, so your controller can use it without defining a separate interface
3. **Export validation middleware** using the `validate` factory function that already exists in the file

The `:id` route parameter also needs validation, but you do not need to create anything new — the existing `validateNumericId` middleware already handles coercing string params to positive integers. You will reuse it when wiring up your routes.

Requirement 5: Create CRUD Endpoints

> **Reference**
>
> See sections 5–6 of the Prisma ORM guide for `findMany`, `create`, `update`, and `delete` operations.

Now build the four endpoints from the table above. Follow the existing file organization pattern — a new route file and a new controller file, both in the `v2/` directories.

Files to create:

- `src/controllers/v2/sayings.ts` — four handler functions, one per endpoint. Each function uses Prisma Client to interact with the database (`findMany`, `create`, `update`, `delete`). Look at the existing `messages` controller for the pattern.
- `src/routes/v2/sayings.ts` — a router that maps HTTP methods and paths to your handler functions. Use the validation middleware you created in Requirement 4 — apply your body validation middleware to POST and PUT routes, and `validateNumericId` to any route with an `:id` parameter. Look at the existing `messages` route file to see how middleware functions are placed in the route chain between the path and the handler. Think about the PUT route — it needs both `validateNumericId` and your body validation. Does the order of those two middleware functions matter? Why or why not?

File to edit:

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

Error handling: For PUT and DELETE, catch the Prisma `P2025` error code (record not found) and return a `404` response. Look at how the existing `messages` controller handles this — the pattern is the same.

Requirement 6: Document Your Routes

> **Reference**
>
> See the OpenAPI Documentation guide for YAML syntax and schema definitions.

Add OpenAPI documentation for all four new endpoints in `openapi.yaml`. Include:

- Path and HTTP method for each endpoint
- A description of what each endpoint does
- Parameter definitions (path params, query params)
- Request body schema for POST and PUT
- Response schemas including error responses

After updating the file, restart the dev server and verify your routes appear in the interactive docs at `http://localhost:3000/api-docs`.

Requirement 7: Write Tests

> **Reference**
>
> See the API Testing guide for Jest + Supertest patterns and mocking strategies.

Create a test file at `tests/v2/sayings.test.ts` (or matching your model name). Follow the mocking pattern from `tests/v2/messages.test.ts` — mock the Prisma client and test each endpoint.

Your tests should cover:

- **GET** returns a paginated list
- **POST** creates a record and returns `201`
- **POST** with an invalid body returns `400` with Zod error details
- **PUT** updates a record
- **PUT** on a non-existent `id` returns `404`
- **DELETE** deletes a record
- **DELETE** on a non-existent `id` returns `404`

```
npm test
```

Your total test count must be **higher than 57** (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 existing v2 routes respond (test `GET /v2/users`)	☐
2	A new model exists in `prisma/schema.prisma` with an `id` and at least one string field	☐
3	A migration file exists in `prisma/migrations/` for the new model (student can show the generated SQL)	☐
4	Seed data exists — `GET /v2/sayings` (or equivalent) returns at least 10 records	☐
5	`POST /v2/sayings` with a valid body creates a new record and returns `201`	☐
6	`POST /v2/sayings` with an invalid body (e.g., empty or missing field) returns `400` with Zod validation errors	☐
7	`PUT /v2/sayings/:id` with a valid `id` updates the record	☐
8	`PUT /v2/sayings/:id` with a non-existent `id` returns `404`	☐
9	`DELETE /v2/sayings/:id` deletes the record	☐
10	All four routes appear in the API docs at `/api-docs` with request/response schemas	☐
11	`npm test` passes and the total test count is higher than 57	☐

#	Verify	✓
12	Student can explain the pipeline: schema → migration → seed → validation → route → controller	☐

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 3: Prisma Intro

Guide Reference

Guide	What It Covers
Prisma ORM	Schema definition, migrations, seeding, Prisma Client CRUD, error codes
Error Handling & Validation	Zod schemas, validation middleware, error responses
Routing & Middleware	Express Router, file organization, middleware chains
API Testing	Jest + Supertest, mocking Prisma, test patterns
OpenAPI Documentation	Documenting routes with OpenAPI/YAML

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

You are welcome to use AI coding assistants to help you write your Prisma model, seed data, validation schemas, routes, controllers, tests, and OpenAPI documentation. However, your checker will ask you to explain the full pipeline — from schema to working endpoint. If you cannot explain what a migration does, how Zod validation works, or how your controller uses Prisma Client, 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.