guide # tooling

OpenAPI Documentation

TCSS 460 — Client/Server Programming

The **OpenAPI Specification** is a standard format for describing a REST API — every endpoint, every parameter, every response — in a single machine-readable document. **Scalar** is the interactive viewer that turns that document into a live documentation page where you and your teammates (and the team building the front-end against your API) can read endpoint descriptions and send real requests. Together, they give your API a front door.

---

1 What is OpenAPI?

The OpenAPI Specification is a YAML or JSON document that describes your REST API in a structured, machine-readable way. It answers three questions about every endpoint:

- What does this endpoint accept? (parameters, request body, headers)
- What does it return? (response shapes, status codes)
- What does it mean? (summary, description, tags)

Any tool that understands OpenAPI can consume that document — a documentation viewer, a code generator, a testing tool, or an AI agent. You write the spec once and everything downstream reads it.

1.1 A Brief History

The terms "Swagger" and "OpenAPI" get used interchangeably, but they have a specific relationship:

Year	What Happened
2011	**Swagger** created — a specification format and toolset for REST API documentation

Year	What Happened
2015	Swagger donated to the Linux Foundation and renamed the **OpenAPI Specification**
2017	**OpenAPI 3.0** released — major restructuring of the spec format
2021	**OpenAPI 3.1** released — aligned with JSON Schema 2020-12

Today, "OpenAPI" refers to the **specification** and "Swagger" refers to the **tools** that predate the rename. When someone says "add Swagger to your API," they mean "describe your API with OpenAPI and serve it with a viewer."

This course uses **OpenAPI 3.0.3** — the version in your starter repo's `openapi.yaml`. It is widely supported across every tool you will encounter.

1.2 The Java Comparison

If you have written Java, you have probably seen Javadoc — structured comments above methods that a tool turns into HTML documentation. OpenAPI does the same thing for HTTP endpoints instead of Java methods.

In Java, you document a method like this:

```java
/**
 * Returns user details for the given ID.
 *
 * @param id The user's unique identifier
 * @return A User object containing name and email
 */
public User getUserById(int id) {
    return userRepository.findById(id);
}
```

In OpenAPI, you document an HTTP endpoint like this:

```yaml
/v1/users/{id}:
  get:
    summary: Get a user by ID
    description: Returns user details for the given ID.
    parameters:
      - in: path
        name: id
        required: true
        schema:
          type: integer
```

```
        description: The user's unique identifier
    responses:
      '200':
        description: User details
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/User'
```

Both are structured documentation that tools transform into something useful. The difference is that Javadoc describes method signatures; OpenAPI describes HTTP endpoints — paths, methods, status codes, and JSON shapes.

> **Machine-Readable Documentation**
>
> Javadoc produces HTML that humans read. OpenAPI produces a structured document that humans **and tools** read. This is what makes it powerful: the same spec that renders your documentation page also drives code generators, API clients, and AI agents.

---

2 Why Document Your API?

You might be thinking: "I know what my endpoints do — why write a separate document?" Here are four reasons that matter specifically in this course and in industry.

2.1 Another Group Builds Against Your API

In TCSS 460, your group builds the back-end during Weeks 2-5. In Weeks 6-10, **another group** builds a front-end that consumes your API. They have never seen your code, and they are not going to read it. Your docs at `/api-docs` are their primary reference — the only window into your back-end.

If your docs are incomplete, they are stuck. If your docs are wrong, they build against a lie and discover the mismatch at the worst possible time. If your docs are missing, your API effectively does not exist for them.

2.2 Documentation as a Contract

Your OpenAPI spec is a **contract** between your team and whoever consumes your API. It says: "If you send this request, you will receive this response." Both teams can develop independently against that contract without coordinating every change.

This is how every professional API works. Stripe, GitHub, and Twilio all publish OpenAPI specs so that developers can build integrations without reading a line of their source code. The spec is the interface. The source code is an implementation detail.

2.3 Scalar Is Your Testing Interface Too

Scalar is not just for the team consuming your API — it is for you during development. You can open `/api-docs`, click any endpoint, fill in the parameters, and send a live request to your running server. It is a built-in API client, always up to date with your documentation.

The consuming team will use it to understand your API. You will use it to verify that your endpoints actually behave the way you documented them.

2.4 AI Agents Need Your Docs

> **Gen AI & Learning: Documentation as Context Engineering**
>
> Your OpenAPI spec is one of the most powerful forms of **context engineering** — giving AI coding tools the right information to produce accurate output.
>
> When you paste your spec into an AI coding agent and ask it to generate a TypeScript API client, the agent reads your endpoint descriptions, parameter types, and response schemas to produce correctly typed interfaces and fetch calls. When you ask it to generate integration tests, it writes tests for the status codes and response shapes you documented.
>
> Better docs = better generated code. If your spec says a response returns `{ id: number, title: string, year: number }`, the agent generates a correctly typed `Movie` interface. If your spec is missing or vague, the agent guesses — and guesses wrong. In Weeks 7-10, you will generate front-end code from your spec. The teams with thorough documentation will have a measurable advantage.

---

3 Your Starter Repo Setup

Your starter repo (`TCSS460-backend-1`) already has the documentation infrastructure wired up. You do not configure anything — you just edit the YAML file and reload the page.

3.1 How It Works

In `src/app.ts`, four lines handle everything:

```
import fs from 'fs';
import YAML from 'yaml';
import { apiReference } from '@scalar/express-api-reference';

// OpenAPI documentation
const specFile = fs.readFileSync('./openapi.yaml', 'utf8');
const spec = YAML.parse(specFile);
app.get('/openapi.json', (_request: Request, response: Response) => {
    response.json(spec);
});
app.use('/api-docs', apiReference({ spec: { url: '/openapi.json' } }));
```

Here is what each part does:

Line	What It Does
`fs.readFileSync('./openapi.yaml', 'utf8')`	Reads `openapi.yaml` from the project root when the server starts
`YAML.parse(specFile)`	Parses the YAML text into a JavaScript object
`app.get('/openapi.json', ...)`	Serves the parsed spec as JSON at `/openapi.json`
`app.use('/api-docs', apiReference(...))`	Mounts Scalar at `/api-docs`, pointing it at your JSON endpoint

The flow is: `openapi.yaml` → parsed at startup → served as JSON at `/openapi.json` → Scalar fetches it and renders the interactive documentation at `/api-docs`.

Viewing Your Docs

Start your server with `npm run dev`, then open `http://localhost:3000/api-docs` in your browser. You should see Scalar rendering the endpoints defined in `openapi.yaml`.

3.2 Why Hand-Maintained YAML?

The most common question: "Why not generate the spec from annotations in the TypeScript files, like swagger-jsdoc?"

The honest answer: there is no currently maintained annotation-to-spec tool for plain Express + TypeScript. `swagger-jsdoc` hasn't been meaningfully updated since ~2022. More opinionated alternatives — `tsoa`, `zod-to-openapi` — require restructuring the entire project around their conventions.

For this course, editing `openapi.yaml` directly is the right approach. It has a genuine pedagogical upside: you read and write the actual OpenAPI specification, not a layer of abstraction on top of it. You learn what the spec says, what the fields mean, and what the format requires. That knowledge transfers to every tool and every project. Annotations teach you what the tool accepts; YAML teaches you what OpenAPI is.

> **The Restart Requirement**
>
> The YAML file is read once at server startup. If you edit `openapi.yaml`, you need to restart your server (or let `npm run dev` handle it via `nodemon`) for the changes to appear in Scalar. This is different from your TypeScript source files, which are recompiled automatically.

---

4 Structure of `openapi.yaml`

Your starter repo includes a populated `openapi.yaml` at the project root. It demonstrates all the structures you will use. Let us walk through each top-level section.

4.1 The Top-Level Sections

```yaml
openapi: 3.0.3
info:
  title: TCSS 460 Backend Example API
  description: An example Express/TypeScript API for TCSS 460.
  version: 1.0.0

servers:
  - url: http://localhost:3000
    description: Local development

tags:
  - name: Hello (v1)
    description: Basic HTTP method demos

paths:
  /v1/hello:
    get:
      tags: [Hello (v1)]
      summary: GET hello
```

```
      # ...

components:
  schemas:
    HelloResponse:
      type: object
      properties:
        message:
          type: string
          example: 'Hello, you sent a GET request'
```

Section	Purpose
`openapi`	The spec version — always `3.0.3` in this course
`info`	API title, description, and version — shown at the top of the Scalar page
`servers`	Base URLs for the API — used by Scalar when you send live test requests
`tags`	Named groups for organizing endpoints in the sidebar
`paths`	All of your endpoint definitions — the main body of the spec
`components`	Reusable schemas, security schemes, and other shared definitions

4.2 Components and `$ref`

The `components` section is where you define schemas that get reused across multiple endpoints. Instead of repeating the full shape of a `Movie` object in five different response definitions, you define it once under `components/schemas` and reference it everywhere else with `$ref`.

```
components:
  schemas:
    Movie:
      type: object
      properties:
        id:
          type: integer
          example: 42
        title:
          type: string
          example: 'The Matrix'
        year:
          type: integer
          example: 1999
```

A `$ref` in a path definition looks like this:

```yaml
responses:
  '200':
    description: The requested movie
    content:
      application/json:
        schema:
          $ref: '#/components/schemas/Movie'
```

The `#` means "this document." The full path `#/components/schemas/Movie` navigates the YAML tree to the `Movie` schema you defined. Change `Movie` in `components` once and every endpoint that references it is updated automatically.

> **Define Your Schemas Early**
>
> In the group project, your endpoints will share resource shapes — books, movies, users, whatever your domain is. Define those as component schemas from the start. It reduces repetition, keeps your documentation consistent, and makes updates trivial.

4.3 You Are Adding to an Existing File

Unlike setting up Swagger from scratch, you are **not** creating `openapi.yaml` — it already exists and already has several endpoints documented. Your job is to add new entries to `paths` and new schemas to `components/schemas` as you build out your API.

The existing entries are working examples. Read them. They show you exactly the format and indentation the file expects.

---

5 Documenting a New Endpoint

This is the section you will come back to most often. Let us walk through every piece of an endpoint definition so you can document any route you write.

5.1 Path and Method

Every endpoint starts with its path nested under `paths`, followed by the HTTP method:

```yaml
paths:
  /v1/movies:
```

```
    get:
      # GET /v1/movies definition goes here
    post:
      # POST /v1/movies definition goes here

  /v1/movies/{id}:
    get:
      # GET /v1/movies/:id definition goes here
```

One path can have multiple methods. Each method gets its own definition block.

5.2 Summary, Description, and Tags

```
get:
  tags: [Movies]
  summary: List all movies
  description: Returns a paginated list of all movies in the catalog. Supports
filtering by genre and sorting by release year.
```

Field	What It Does
`tags`	Groups this endpoint with others in the Scalar sidebar
`summary`	The one-line description shown in the collapsed endpoint list
`description`	A longer explanation shown when you expand the endpoint

`summary` is required. `description` is optional but recommended for anything non-obvious. `tags` groups your endpoints by resource — one tag per resource type is a good convention.

5.3 Path Parameters

Path parameters are the dynamic segments of a URL — what Express calls `:id`. In OpenAPI, you use curly braces: `{id}`.

```
paths:
  /v1/movies/{id}:
    get:
      tags: [Movies]
      summary: Get a movie by ID
      parameters:
        - in: path
          name: id
          required: true
          schema:
```

```
          type: integer
        description: The movie's unique numeric ID
```

> **The Most Common Mistake: `:id` vs `{id}`**
>
> Express uses `:id` in route definitions. OpenAPI uses `{id}`. They mean the same thing, but the syntax is different. If you write `:id` in your OpenAPI path, Scalar will not recognize it as a parameter and the "Try It" feature will not prompt for the value.
>
> Express route: `router.get('/movies/:id', ...)`
>
> OpenAPI path: `/v1/movies/{id}`

Path parameters are always `required: true`. If the value is not in the URL, the route does not match.

5.4 Query Parameters

Query parameters come after the `?` in a URL. Use `in: query` and omit `required` (it defaults to `false` for optional parameters):

```yaml
get:
  tags: [Movies]
  summary: List all movies
  parameters:
    - in: query
      name: genre
      schema:
        type: string
      description: Filter by genre (e.g., "Sci-Fi", "Drama")
    - in: query
      name: limit
      schema:
        type: integer
        default: 20
      description: Maximum number of results to return
    - in: query
      name: offset
      schema:
        type: integer
        default: 0
      description: Number of results to skip (for pagination)
```

The `default` field tells readers what value your handler uses when the parameter is not provided. Include it for any parameter that has a meaningful default.

5.5 Request Body

POST and PUT endpoints accept data in the request body. Use the `requestBody` field:

```yaml
post:
  tags: [Movies]
  summary: Add a new movie
  requestBody:
    required: true
    content:
      application/json:
        schema:
          type: object
          required:
            - title
            - year
          properties:
            title:
              type: string
              example: 'Dune: Part Two'
            year:
              type: integer
              example: 2024
            genre:
              type: string
              example: 'Sci-Fi'
```

The `required` array inside `schema` lists which fields must be present. Fields not listed are optional. The `example` values appear as pre-filled sample data in Scalar's request editor, which makes it much easier for the consuming team to understand what to send.

5.6 Responses

Document every status code your handler returns. This is the part developers most often under-document — they write the `200` and forget the `400`, `404`, and `500`. The consuming team needs all of them.

```yaml
responses:
  '200':
    description: The requested movie
    content:
      application/json:
        schema:
          $ref: '#/components/schemas/Movie'
  '400':
    description: Invalid movie ID format
    content:
      application/json:
        schema:
          $ref: '#/components/schemas/ErrorResponse'
  '404':
    description: Movie not found
    content:
      application/json:
```

```
        schema:
          $ref: '#/components/schemas/ErrorResponse'
  '500':
    description: Internal server error
    content:
      application/json:
        schema:
          $ref: '#/components/schemas/ErrorResponse'
```

Notice that all the error responses reference a shared `ErrorResponse` schema — which you define once in `components/schemas` and reuse everywhere.

> **Document Every Status Code You Return**
>
> Every `response.status(N).json(...)` in your route handler should have a corresponding entry in your OpenAPI definition. If your handler can return 200, 400, 404, and 500, document all four. An undocumented status code is invisible to the consuming team — and to AI tools generating code from your spec.

5.7 Adding a Reusable Schema

When you need a new resource shape, add it to `components/schemas`:

```
components:
  schemas:
    Movie:
      type: object
      properties:
        id:
          type: integer
          example: 42
        title:
          type: string
          example: 'The Matrix'
        year:
          type: integer
          example: 1999
        genre:
          type: string
          example: 'Sci-Fi'

    ErrorResponse:
      type: object
      properties:
        error:
          type: string
          example: 'Movie with ID 42 not found'
```

Then reference it from any path definition with `$ref: '#/components/schemas/Movie'`.

Try It Yourself

Document a complete endpoint from scratch:

1. Open `openapi.yaml` in your starter repo
2. Add a new schema to `components/schemas` — define an object with at least three properties
3. Add a new path under `paths` — include at least one path parameter, a `200` response using `$ref`, and a `404` response
4. Restart your dev server and open `http://localhost:3000/api-docs`
5. Verify your new endpoint appears in the Scalar sidebar under the correct tag
6. Click into it and confirm the parameter and response schema render correctly

Gen AI & Learning: Generating YAML from Existing Code

AI coding agents can draft OpenAPI YAML from your route handler code. Select a handler function and ask your agent to "write the OpenAPI YAML entry for this endpoint." It will read the parameters, request body destructuring, response status codes, and JSON shapes to produce a starting draft.

Always verify the output. The agent infers types from your code and can get them wrong — especially for optional fields, error conditions, and edge cases you handle implicitly. Check the generated YAML against your actual handler, then add any missing status codes and fix any type mismatches. The agent produces a first draft; you own the final spec.

---

6 Testing from Scalar

Scalar includes a built-in API client. You can send real requests to your running server directly from the documentation page — no Postman required during development.

6.1 Using the Built-In Client

1. Open `http://localhost:3000/api-docs` with your server running
2. Click any endpoint in the sidebar to expand it
3. Review the documented parameters and response schemas
4. Click **Try** (or the equivalent button in Scalar's UI)

5. Fill in any required parameters or request body fields
6. Send the request and see the live response from your server

This informal manual testing is useful during development — it is faster than switching to a separate tool for a quick check. It is different from the automated test suite (Jest/Supertest) that runs against your entire API; Scalar is for ad-hoc exploration.

6.2 Using Scalar to Verify Your Docs

Scalar's test client serves a purpose beyond convenience: it is how you confirm that your documentation matches your actual implementation.

When you send a request through Scalar and get back a response, compare it to the schema you documented. If your docs say the `200` response includes `{ id, title, year }` but your handler returns `{ id, title, releaseYear }`, you have a mismatch. Fix either the code or the docs — but fix it before the consuming team builds against the wrong field name.

Try It Yourself

1. Open `/api-docs` with your server running
2. Find the `GET /v1/hello` endpoint and expand it
3. Use the built-in client to send the request
4. Compare the actual response fields to the schema defined in `openapi.yaml`
5. Now open `openapi.yaml` and add a field to the `HelloResponse` schema that your handler does **not** return
6. Restart and send again — notice the documented shape no longer matches the actual response
7. Remove the phantom field and confirm they match again

6.3 The Consuming Team's Perspective

During the cross-group swap, the front-end team will visit your `/api-docs` URL and do exactly what you just did — read the endpoint descriptions, inspect the schemas, and use the built-in client to understand what your API returns before writing a single line of front-end code. Your documentation is their onboarding experience.

---

7 Keeping Docs in Sync

Documentation drift — when code changes but docs do not — is the most common documentation problem. It produces two bad outcomes: the consuming team builds against a spec that no longer matches reality, and AI tools generate code based on stale information.

7.1 Write Docs as You Write Routes

The best time to document an endpoint is while you are writing the route handler — you already know the parameters, request body, and response shapes because you just wrote the code. If you wait until the end of a sprint, you will forget the details that matter: which fields are optional, what the 400 response looks like, what edge cases return 404 vs 422.

Some developers write the YAML entry first, as a form of design — "what should this endpoint accept and return?" — and then implement the handler to match. This is legitimate and can help catch design problems before they become code problems.

7.2 Docs Are the Source of Truth for the Consuming Team

The front-end team reads your docs, not your code. This has concrete implications:

- If your docs say a field is called `movieId`, the front-end team sends `movieId` — even if your handler internally uses `movie_id`
- If your docs say a `POST` requires `title` and `year`, they send `title` and `year` — even if your handler also silently accepts `genre`
- If an endpoint is not documented, it does not exist for them

Your docs define the surface of your API, not your code. If they diverge, the divergence is a bug — and the question of which one is "right" depends on which came first.

Gen AI & Learning: Wrong Docs Mean Wrong Generated Code

AI tools generate client code from your OpenAPI spec. They follow the spec, not your source code. If your spec says a response includes `movieTitle` but your handler actually returns `title`, the AI generates a client that reads from `movieTitle` — and that code silently returns `undefined` at runtime.

This is why documentation accuracy matters more in an AI-assisted workflow than in a traditional one. A human developer can look at a network response and notice the field name mismatch. An AI agent cannot. It trusts the spec. If the spec is wrong, the generated code is wrong, and the bug is invisible until something breaks.

Keep your spec accurate. The AI is only as reliable as the context you give it.

---

8 Summary

Concept	Key Point
OpenAPI Specification	A standard YAML/JSON format describing every endpoint, parameter, and response in your API
Scalar	The interactive documentation viewer served at `/api-docs` — reads your spec and renders it
`openapi.yaml`	A standalone file at your project root — you edit this directly to document your API
`paths`	The main section of the spec — one entry per URL path, one block per HTTP method
`components/schemas`	Reusable shape definitions — define once, reference with `$ref` everywhere
`$ref`	A reference to a schema defined in `components` — keeps your spec DRY
Path parameters	Use `{id}` in OpenAPI paths (not `:id` like Express) — the most common mistake

Concept	Key Point
Query parameters	Use `in: query`, optional by default, document `default` values when relevant
`requestBody`	Documents POST/PUT request bodies — include `required` fields and `example` values
Responses	Document every status code your handler can return — 200, 400, 404, 500, all of them
Tags	Group related endpoints in the Scalar sidebar — one tag per resource is a good convention
Docs as contract	Your spec is the source of truth for the consuming team — accuracy is not optional
Scalar test client	Built-in API client for informal manual testing during development

---

9 References

Official Documentation:

- OpenAPI Specification 3.0.3 — The full specification reference
- OpenAPI Specification Overview — Swagger's annotated guide to the spec
- Scalar Documentation — Official Scalar docs, themes, and configuration
- `@scalar/express-api-reference` on npm — The package that mounts Scalar in Express
- `yaml` package on npm — The YAML parser used to read `openapi.yaml` at startup

---

10 Further Reading

External Resources

- OpenAPI Initiative — The organization maintaining the OpenAPI standard
- Swagger Editor — Online editor for writing and validating OpenAPI YAML with live preview
- OpenAPI Generator — Generate TypeScript clients, server stubs, and more from a spec
- OpenAPI Map — Visual reference of the OpenAPI 3.0 object model
- API Design Patterns — Designing Web APIs (Jin, Sahni, Shevat, 2018) — O'Reilly book on API design principles

---

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