lectures # week-3

Handling Input & the Proxy Pattern (Tuesday, April 14, 2026)

Lecture Recording

Watch on Panopto

Today's Agenda

1. **Course admin** — Sprint 1, Week 3 check-off, reading load expectations
2. **Framing** — you are the back-end team; your partner group will build the front-end against your API
3. **OpenAPI & Scalar** — what the `/docs` route is doing and why it matters
4. **Four ways input gets into a handler** — query strings, route params, body, headers
5. **V1 controllers** — pulling input off the `request` object with destructuring
6. **V2 validation middleware** — moving input validation out of controllers
7. **Proxy routes** — pass-through vs. transform proxies, server-as-client, `fetch` and `await`

Plan & Admin 0:00

What You Should Be Working On

- **Sprint 1 (group)** — you should be *hot and heavy* on Sprint 1. By end of week your group should have an alpha release posted with endpoints working. A checklist for next week's sprint review will be posted in Discord — next Tuesday's meeting will hit your live API on Postman.
- **Proxy API check-off (individual)** — do this tonight or tomorrow, even if it doesn't get peer-checked-off right away. The individual work is *adjacent* to the sprint: the check-off builds proxy routes against the weather API while Sprint 1 builds proxy routes against

TMDB. Getting the check-off mechanics under your belt first makes the group work much easier.

- **Week 3 reading + quiz** — the concept reading is about relational databases. Prisma is covered in the guides, not in this week's concept reading.

Reading Load Expectation

- **Concept readings** will stay about 4,000–5,000 words — plan **~1 hour** to read carefully, not 30 minutes.
- **Guides are reference material** — *not* mandatory cover-to-cover reading. Skim for the big ideas, then come back to them when you're actually implementing. There is more information in the guides than lecture could ever cover, even with more lecture time.
- **Pacing** — the course front-loads everything you need for the API in the first ~2 weeks so you can build something real, then front-loads front-end material in Weeks 6–7. It has to ramp fast to give you time to do cool stuff.

Back-end Team / Front-end Team Framing 2:40

For the next three weeks, pretend your group is **the back-end team at a company**. Later in the quarter you'll pivot and build a *front-end* — but not for your own API. At the end of Sprint 4 you'll hand your running API (the live deployment plus the OpenAPI docs, not the source code) off to a different group, and *they'll* build a front-end on top of it. In turn, *you* will receive a back-end handoff from yet another group and build your front-end on theirs.

Nine groups, nine such pairings, arranged in a ring. On each edge of the ring, the two groups are still "the same company" — not a third-party integration, just back-end and front-end teams under one roof. But across the quarter, your group sits in two different companies: one where you're the back-end team, one where you're the front-end team.

The picture of a user searching for a movie:

1. **Front-end** sends a request to **your API** (the other group's front-end hits your back-end)
2. **Your API** proxies the request to **TMDB**, gets data back, and may transform it
3. **Your API** sends the transformed response to the front-end
4. **Front-end has no idea TMDB exists** — as far as it knows, your API is the source of truth

That's what "proxy routes" means in this course: your back-end sits between the front-end and the third-party API. The front-end never talks to TMDB directly.

Rock-climbing site analogy

Adding weather to a movie app doesn't make much sense — but adding weather to a rock-climbing site totally does. The proxy pattern lets you stitch together any third-party data you need under your own API surface.

OpenAPI & Scalar Documentation 10:21

Swagger → OpenAPI

Swagger came out about 15 years ago as an attempt to standardize API documentation so every API's docs would look and feel the same. It took off, and its underlying schema was formalized into the **OpenAPI Specification** — a structured description of your API (routes, parameters, responses, error codes) that tooling can consume.

The OpenAPI spec in this course lives in a YAML file alongside your code. Raw YAML is a pain to read and a pain to edit by hand — there are editors that help, and **AI is fantastic at writing and maintaining it for you**. Understand what's in the file, but don't hand-write it from scratch.

Scalar — hosting the docs

Scalar is a module that consumes an OpenAPI YAML file at build time, turns it into HTML, and serves it as a static route on your API. When you `npm run dev`, you get:

- `localhost:3000` — the API itself
- `localhost:3000/docs` — Scalar-rendered documentation of every route

Think of `/docs` like the Oracle Java API docs — it's the reference students in three weeks will rely on to build a front-end against your API.

Try-it-in-the-docs

Scalar lets you fire real requests from the docs page, just like Postman. Pick a route, fill in parameters/body, click send. It hits whatever server served the docs — so `localhost:3000` docs will hit your running dev server.

Four Places Input Comes From 15:16

Last week's Hello-World routes took no input. Real routes take input from one (or more) of four places:

Source	URL/Payload Location	Typical Use
Query string	`?q=foo&limit=10`	Filtering zero-to-many results
Route (path) parameter	`/users/:id`	Addressing a single resource
Body	JSON in the HTTP body	Payload for `POST` / `PUT` / `PATCH`
Headers	HTTP headers	Auth tokens, content type, custom metadata

Query Strings 15:16

Use when: the route returns **zero-to-many** resources and query parameters act as filters.

```
GET /v1/input/search?q=interstellar&limit=10
```

- **Good fit:** "all movies from August 1977" — `?year=1977&month=august`
- **Wrong fit:** "one specific movie by id" — that's a route parameter

Your docs should spell out each parameter: type, whether it's required, valid range. The more you put in the docs, the friendlier your API is for the team consuming it.

Route (Path) Parameters 19:07

Use when: you're addressing **one specific resource**. The URL has *no* `?key=value` pairs — the parameter is embedded directly in the path:

```
GET /v1/input/users/42
```

Your response should be **either**:

- `200` with the resource, **or**
- `404` Not Found

Never return an empty `200` "nothing found." The whole point of `404` is to communicate "no resource at that address."

> **Terminology**
>
> Express calls these "**route parameters**" — but the framework-agnostic term is "**path parameter**." You'll hear both. They mean the same thing.

Request Body 21:20

Use when: you're sending a payload, typically on `POST` / `PUT` / `PATCH`. Body content is JSON (modern web frameworks assume JSON bodies).

```
POST /v1/input/users
Content-Type: application/json

{
  "name": "Jane Doe",
  "email": "jane@example.com"
}
```

Don't mix query strings with a POST body. Technically Express will let you write a handler that reads both — but many front-end HTTP libraries *won't* send a query string with a POST. Be idiomatic: bodies for POST, query strings for GET filters, path params for single-resource GETs.

> **Student Question**
>
> Q: Why is HTTP "human-readable" but we still see escape characters?
>
> A: The *wire format* is text that a human can read, which is why you see quoted strings and escape sequences in curl output — but the real consumer of a web API is **other code**, not a human. Your API responses are JSON, not HTML, because the front-end code will parse the JSON and render something pretty from it. The browser-facing experience is the front-end's job, not the back-end's.

Headers 25:34

Use when: metadata about the request — authorization tokens, content type, custom request IDs. You probably won't write custom headers in this course, but you *will* use

`Authorization: Bearer <jwt>` once auth comes online.

Authentication flow preview (Weeks 4–5):

1. Front-end hits the **auth service**, logs in, gets a token back
2. Front-end stores that token (local storage / session storage / cookie — strategies vary)
3. Every subsequent request to the back-end includes `Authorization: Bearer <token>`
4. A JWT-check middleware verifies the token before the controller runs

> **Student Question**
>
> Q: Where will we store the JWT on the client side?
>
> A: We'll decide in the front-end weeks. Trade-offs: do you want users to re-login when they close a tab? Keep a long session across browser restarts? The storage choice (local storage vs. session storage vs. cookies) encodes that answer.

V1 Controllers — Reading Input 27:23

The V1 API in the demo uses **no middleware** — controllers do everything. That makes it easy to see how each input source is pulled off the `request` object.

Query Strings — `request.query`

```
const searchByQuery = (request: Request, response: Response) => {
    const { q, limit } = request.query;  // destructuring
    // ...
};
```

`request.query` is a JavaScript object representing every query-string key/value pair. **Object destructuring** on line with `const { q, limit }` is idiomatic JavaScript — it's equivalent to:

```
const q = request.query.q;
const limit = request.query.limit;
```

but in one line. You'll see destructuring everywhere in modern JS/TS code.

Route Parameters — `request.params`

```
const getUserById = (request: Request, response: Response) => {
    const { id } = request.params;
    // ...
};
```

Exact same pattern — just a different property on `request`.

Body — `request.body`

```
const createUser = (request: Request, response: Response) => {
    const { name, email } = request.body;
    // ...
};
```

Same pattern as `query` and `params` — **but** this only works because of a middleware function registered in `app.ts`:

```
// app.ts
import express from 'express';
import cors from 'cors';

const app = express();

app.use(cors());
app.use(express.json());   // ← parses JSON bodies into JavaScript objects
app.use(logger);

// ... routes mounted here
```

Without `app.use(express.json())`, `request.body` would be the raw string coming off the HTTP wire — not a parsed object. The next section walks through what that middleware is doing on every request.

Headers — bracket notation

```
const readHeaders = (request: Request, response: Response) => {
    const xRequestId = request.headers['x-request-id'];
    const contentType = request.headers['content-type'];
    const auth = request.headers['authorization'];
    // ...
};
```

Header names are looked up with **bracket notation and string keys**, because most header names contain hyphens (`content-type`, `x-request-id`) that aren't valid JavaScript identifiers.

> **Lecture Correction — Dot Notation on Headers**
>
> In lecture I said you "can't access the headers using the dot notation." That's only true for hyphenated names — a header like `authorization` works fine as `request.headers.authorization`. The *reason* we default to bracket notation is that most headers (`content-type`, `x-request-id`) have hyphens that dot notation can't handle, so bracket notation is the safer, universally-applicable convention.

Why `request.body.xyz` Just Works

Notice that we're writing `request.body.name`, `request.params.id`, `request.query.q` with no type annotations and no interface definitions — and TypeScript doesn't complain, no matter what field name we reach for. Why?

The Express type definitions declare `request.body`, `request.params`, and `request.query` with **intentionally loose types**. Roughly:

- `request.body` is typed as `any`
- `request.params` is typed as `ParamsDictionary` (a record of string keys → string values) that accepts any key
- `request.query` is typed as a nested record of string keys → string/array/nested values

These types are **escape hatches** that mostly disable TypeScript's checking on those objects. That's a deliberate design choice: the shape of those objects **isn't known at compile time**. The data is runtime input from outside your codebase — an HTTP request — so the framework can't give you a precise type up front. You, the developer, are responsible for validating the shape at runtime (which is exactly what the V2 middleware does).

Compare to Java: you'd need a class definition for every request shape — `CreateUserRequest`, `SearchQuery`, `UserIdParam` — and the compiler would check them. TypeScript is a middle ground: strict where it can be, with deliberate escape hatches for boundary data it can't reason about.

> **Coming next week — adding to `request`**
>
> Middleware will sometimes want to *attach* new fields to the request (e.g., a parsed user id after JWT verification) so the controller can read them. TypeScript **does** complain about that — it only gives you a free pass on the fields the library already declared. We'll use a technique called **module augmentation** to extend Express's types and silence that error.

> **Don't use `any` in your own code**
>
> TypeScript gives you `any` and `unknown` as escape hatches. In library type definitions like Express's, they're unavoidable. In **your** code, avoid them — the whole point of using TypeScript is the type checking. Validate at the boundary, type strictly afterwards.

Why `express.json()` Matters 33:37

When an HTTP body arrives, it's a **string** of characters — well-formed JSON text, malformed JSON, XML, or anything else. Without parsing, `request.body` would be that raw string.

`express.json()` is the middleware that:

- **Happy path** — parses a well-formed JSON body into a JavaScript object, assigns it to `request.body`, calls `next()`
- **Sad path** — malformed JSON / XML / unparseable text → sends `400`
- **No body at all** — fine, calls `next()`

In `app.ts`:

```
app.use(cors());
app.use(express.json());   // ← critical for all body-parsing
app.use(logger);
```

Without `express.json()` in the pipeline, **every** controller handling a POST/PUT/PATCH would need its own try/catch JSON parse. With it, every downstream controller can assume `request.body` is already a JavaScript object.

> **The middleware payoff**
>
> This is the first moment the middleware pattern starts paying off. One line in `app.ts` replaces 30 try/catch blocks across 30 controllers.

V2 — Validation Moves to Middleware 38:06

The V2 routes demonstrate the **more idiomatic** pattern: push input validation out of the controller and into middleware functions. Even if a validator is only used once, this **separates responsibilities** — middleware validates the *shape* of input, controller handles the *business logic*.

```
router.get('/search', validateSearchQuery, searchByQueryV2);
router.get('/users/:id', validateNumericId, getUserByIdV2);
router.post('/users', validateUserBody, createUserV2);
```

Statement-Based Validation (this repo)

The Week 3 demo writes validation as plain `if` / `else` middleware functions — no external library. Example:

```
const validateSearchQuery = (
    request: Request,
    response: Response,
    next: NextFunction
) => {
    if (!request.query.q) {
        return response.status(400).json({
            error: 'Missing required parameter: q',
        });
    }
    next();
};
```

Rules Recap

Every middleware must do **exactly one** of:

- **Send an HTTP response** (sad path — validation failed → `400`)
- **Call `next()`** (happy path — move on to the next function)

Every branch must end in one or the other. Not both. Not neither.

> **Coming next week — Zod**
>
> The next demo repo will use **Zod** for schema-based validation. You define the shape you expect, and Zod generates the validator. Much more idiomatic and reusable than hand-written `if` / `else` blocks — but learn the statement-based version first so you understand what Zod is doing under the hood.

Java vs. TypeScript — Why You Validate Types at Runtime 44:00

Consider this validator:

```
const id = Number(request.params.id);
if (!Number.isInteger(id) || id < 0) {
    return response.status(400).json({ error: 'id must be a non-negative
integer' });
}
```

In Java (TCSS 305), you'd never write `if (!isInteger(x))` for a parameter typed `int` — the **compiler** rejects a String argument at compile time. Why do we write the check here?

Because the input isn't TypeScript code — it's a string off the HTTP wire. HTTP is text. Everything in `request.query`, `request.params`, and `request.headers` starts as a **string**, regardless of what the TypeScript types say. The validation converts and checks the runtime value; the compiler can't help because the string is user input, not code.

This is also one of the first times you're writing code where **you are not the consumer**. The front-end team will hit your routes with whatever input their form collected. Validation + good docs are how you communicate the contract.

> ⚠ **Query strings and path params are always strings**
>
> Destructuring `const { limit } = request.query` doesn't give you a `number`, it gives you a `string`. If you need arithmetic on it, convert with `Number(limit)` *and* check `Number.isInteger(...)` — `Number('abc')` returns `NaN`, which is not an integer and will fall into your 400 branch.

Multi-Error Validation Pattern 46:24

When multiple fields could be wrong, **collect all errors and return them together** — don't fail at the first one. This gives front-end developers one round-trip to see everything wrong with the request instead of playing whack-a-mole.

```
const validateUserBody = (
    request: Request,
    response: Response,
    next: NextFunction
) => {
```

```
    const errors: string[] = [];

    const { name, email } = request.body;

    if (!name || typeof name !== 'string') {
        errors.push('name is required and must be a string');
    }
    if (!email || typeof email !== 'string') {
        errors.push('email is required and must be a string');
    }

    if (errors.length > 0) {
        return response.status(400).json({ errors });
    }

    next();
};
```

Document the sad path too. Your Scalar docs should show the `400` response shape so front-end developers know what to expect. Tests should fail if the numbers are wrong — front-end code writes **sad-path handlers** that branch on status codes, and a wrong code breaks those handlers.

> **Happy path vs. sad path, revisited**
>
> Front-ends don't generally branch on `200` vs. `201` — success is success. But they absolutely branch on `400` vs. `404` vs. `500`. Your status codes are part of your API contract.

JavaScript Arrays are Weird 51:56

Coming from Java, expect some surprises:

Behavior	Java	JavaScript
Resize	Arrays are fixed-size (use `ArrayList`)	Arrays are fully mutable — `push` / `pop`, direct index
Negative index	Compiler error	Works, but **doesn't wrap** like Python — it creates an object property, not an array slot

Behavior	Java	JavaScript
Out-of-bounds assignment	`ArrayIndexOutOfBoundsException`	Silently expands the array with `undefined` holes

```
const errors = [];
errors.push('first');           // ['first']
errors[4000] = 'surprise';      // length is now 4001; indices 1–3999 are `undefined`
errors[-1] = 'also weird';      // this is a property, not an array element — length unchanged
```

> ⚠ **Python people**
>
> `errors[-1]` in JavaScript does **not** give you the last element the way Python does. It attaches `-1` as a property key on the array object. If you want the last element, use `errors[errors.length - 1]` or `errors.at(-1)`.

For this course, stick with `push` / `pop` for accumulation and direct indexing when you know the bounds. The weird cases are there if you ever need to debug someone else's code.

Validation Isn't Documentation 55:15

There's a school of thought that **good validation is the spec**. In environments where consumers can read your code, that holds up. In *this* environment — consumers only see your `/docs` page — validation is **not** a substitute for documentation.

That said, good validation *enables* good documentation. If your validator is clear about what's required and what the rules are, writing the OpenAPI YAML (or having AI write it from your validator) becomes mechanical.

Proxy Routes — Pass-Through vs. Transform 56:11

A **proxy route** in your API forwards the request to an external API, then returns the response to the client. Two flavors:

Flavor	Input	Output
Pass-through proxy	Same params as the upstream API	Raw upstream response, unchanged
Transform proxy	Similar to upstream, possibly adjusted	Upstream response **reshaped** before returning

Transform Proxy — Why

Example: TMDB returns a movie with **its own rating system**, but your client wants to build its own rating system. Don't leak TMDB's rating shape to the front-end — strip it out in your proxy, attach your own rating data, send the combined payload.

Which one is "right"? Neither — **it depends** on what the consumer needs. It's the first real "it depends" decision you'll make as API designers.

Two Reasons to Prefer Transform Proxies

1. **Abstraction** — your docs describe **your** API, not TMDB's. If TMDB gets too expensive, gets sunset, or you switch providers, you swap out the transform logic and *your* API surface doesn't change. Front-ends don't know anything changed.
2. **Secrets stay server-side** — TMDB / OpenWeather API keys live in your server's config vars, never in client code. If the front-end called TMDB directly, the key would ship to every browser that loaded your site.

Server Becomes Client — `fetch` and `await` 1:02:07

When your Express controller makes a proxy call, your **server becomes a client** for another server. This chain keeps going:

```
Front-end  ──HTTP──▶  Your API  ──HTTP──▶  TMDB  ──SQL──▶  TMDB's database
(client)   ──────▶  (server/client) ──────▶  (server/client)  ──▶ (server)
```

The fetch call

```
const proxyWeather = async (request: Request, response: Response) => {
    const url = `https://api.openweathermap.org/...?
appid=${process.env.WEATHER_KEY}`;

    const result = await fetch(url);          // HTTP call — await because it
takes time
    const data = await result.json();          // parse body text into a JS
object — also await

    response.status(200).json(data);
};
```

What `fetch` is doing:

- **`fetch(url)`** — sends an HTTP GET to another server and returns a `Response` object
- **`await`** — the call is network-bound and slow; `await` lets the Node event loop service other requests while we wait, instead of blocking
- **`result.json()`** — the HTTP body is always text. `.json()` parses the text into a JavaScript object. Also async.

> **fetch vs. axios**
>
> `fetch` is native to Node (no extra module). On the back-end, `fetch` is idiomatic. On the front-end, **Axios** is more common. Axios works on both, but keep the conventions: `fetch` server-side, `axios` client-side.

What To Do This Week 1:05:15

- **Check-off (individual):** build more proxy routes against the OpenWeather API. Explore the weather API's docs.
- **Sprint 1 (group):** build proxy routes against TMDB. Before you write any handler code, **spend one to four hours as a group reading TMDB's docs** — what endpoints exist, what shape do responses have, what does your API need to transform. You cannot write a proxy until you know what you're proxying.
- **Quiz 3 + concept reading** — relational databases, prep for next week.

In the Guides

- The Proxy Pattern — full walkthrough with the weather API
- Routing & Middleware — middleware pipeline, validation patterns
- OpenAPI Documentation — authoring the YAML, Scalar integration
- Intro to Express — Handling Input — query, params, body, headers

---

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