lectures # week-1

What is an API? From Java to the Web (Thursday, April 2, 2026)

Lecture Recording

Watch on Panopto

Today's Agenda

1. **Course admin** — new guides, lecture recaps, readings & quizzes, check-off preview
2. **Group project** — signup, Sprint 0 plan, tech stack, client conversation
3. **What is an API?** — Java `List` interface as analogy, the car analogy, documentation
4. **What is a web API?** — Internet vs. WWW, HTTP protocol, why web APIs exist
5. **Anatomy of a web API request** — URLs, localhost, ports, routes, parameters, HTTP methods
6. **HTTP responses** — status codes, response body, JSON

Plan & Admin 0:03

Brief admin updates, then theory lecture on APIs and web APIs. No code walkthrough today — that moves to next Tuesday.

Course Site Updates 0:25

New Guides Available 0:29

All **TypeScript Fundamentals** and **Tools** guides are complete. New this week — all **Back-End Web API** guides are finished:

- Intro to Express — the framework we're using to build our web API
- Routing & Middleware — probably not covered today, but read ahead for Tuesday
- Error Handling & Validation
- API Testing Tools — the tools you'll see demonstrated in class
- OpenAPI Documentation — documentation structure for your web API

> **In the Guides**
>
> Read these guides multiple times. The more you absorb now, the more we can do later. The guides go deeper than lecture — use them alongside the check-off assignments.

Lecture Recaps 1:18

Tuesday's lecture recap is posted. Every lecture gets a timestamped recap — if you missed class, read the recap and click timestamps to jump to specific points in the Panopto recording.

Readings & Quizzes 2:07

Week 1 concept readings (Networking Fundamentals and HTTP & the Web) should be read before today's lecture. Today's theory content glosses over what those readings cover in detail. Canvas quizzes are based on the concept readings, not the guides.

> **Student Question**
>
> Q: Where are the quizzes — under Concepts or Guides?
>
> A: Quizzes cover the **Concepts** tab (theory, language-agnostic). The **Guides** tab is language-specific how-to material (e.g., "build a route in Express"). Concepts explain *what and why*; guides explain *how*. The two cross-reference each other.

Check-Off 1 & AI Tutor 5:45

Check-Off 1 is due next Sunday (not this Sunday), but you can start it now. You'll clone the lecture demo repo — the same code we'll walk through on Tuesday.

The check-off page has an **AI tutor** chat widget (the little chat bubble). This is a prototype knowledgeable AI tutor that has context about the specific assignment. It uses Socratic method — it won't give you the answer directly, but will ask pointed questions to guide you.

- Chats are recorded but **completely anonymous** — no IP tracking, no identifying information
- There's a 50-message limit (don't circumvent it — it comes out of the instructor's pocket)
- Try to break the guardrails if you want — that's useful feedback for improving it

---

Group Project 12:35

Group Signup 12:42

Sign up for groups on Canvas by Friday night. Anyone not in a group by the weekend will be assigned. Groups will be rebuilt as Group 1–9 (no cheeky names) to align with assignment infrastructure.

Sprint 0 Plan 13:32

Sprint requirements will be released on Thursdays (the week before the sprint is due), so you can do sprint planning during your current sprint. Sprint 0 instructions will go out Sunday after groups are finalized.

Sprint 0 scope:

- Form the group and have an introductory meeting
- Everyone works on their own branch in the same repo — create a made-up route, merge to main, deal with the merge conflicts
- Have a starter project with a single hello-world endpoint
- Deploy that starter project to the cloud

Tech Stack & Architecture 15:51

The Group Project Overview describes the full stack:

- **PostgreSQL** — relational database for user ratings and reviews

- **Prisma** — ORM sitting between your TypeScript code and the database (no raw SQL this quarter)
- **Express + TypeScript** — your web API, which proxies to TMDB (The Movie Database)
- **Next.js + React** — frontend built by another group on top of your API
- **Auth** — provided by an instructor-hosted OAuth provider

All groups will build support for both movies and TV shows (changed from the original even/odd split).

Client Conversation 17:56

The project requirements live in a Client Conversation — a fictional transcript between a non-technical client and their tech-savvy friend. The requirements are **intentionally vague**. The client says things like "I need to be able to search for a movie" — they never say "create a GET endpoint called `/search` with pagination."

This is by design:

1. **Natural variance** — different groups will design different APIs, which makes the frontend swap meaningful
2. **Real-world practice** — clients don't hand you an interface specification; you design the API

Grading & Demos 25:31

The group project is 50% of the course grade, graded with **broad strokes**. Starting Week 3, one class day per week will be group meeting time where each group demos their sprint MVP to the instructor. If you're supposed to have something working and it's not, that's a problem — but the instructor is not nitpicking code quality.

Student Question

Q: What about security — SQL injection, rate limiting, brute force attacks?

A: Modern ORMs like Prisma handle SQL injection protection for you. Rate limiting and other security measures will be **discussed in lecture** but are **not required in your implementation**. The auth system is provided, not built by you. Security is important but there's only so much we can cover in ten weeks — the secure programming course (TCSS 480) is specifically designed for that.

Database Scope 22:38

Student Question

Q: Is there already a database for the movies?

A: TMDB has all the movie and TV show data — your web API **proxies** to TMDB to get that data. Your own PostgreSQL database stores **user-specific data**: users, ratings, and reviews. You design those tables yourself. When your API proxies to TMDB, you can either return TMDB's response directly or reshape it before sending it to the frontend — but whatever you return must be documented.

What is an API? 29:23

Starting with Java code you already know to build up the concept of an API before moving to web APIs.

The Java `List` as an API 29:40

Consider this Java code:

```
List<String> names = buildNames();
```

What's the type of `names`? It's `List` — an **interface**. You don't know if the underlying implementation is an `ArrayList`, a `LinkedList`, or a custom implementation. And it doesn't matter.

Some call this interface an **ADT** (Abstract Data Type). Others call it the **API** — the Application Programming Interface. Focus on the **I**: this is the **interface** for how you, the client, interact with whatever the API defines.

Key insight: API documentation tells you how to interact with the object. `names.size()` returns an `int` — how do you know? Documentation. `names.getFirst()` exists — how do you know? Documentation. `names.add("Charles")` takes a `String` — how do you know? Documentation. Without documentation, you're guessing method names and hoping the compiler catches your mistakes.

The Car Analogy 34:27

An automobile has an API: the **steering wheel**, the **accelerator pedal** (not "gas pedal" — electric cars don't have gas), and the **brake pedal**. This interface is the same whether you drive a Kia Rio or a Camry. The steering wheel might look different, might have extra buttons, but the core interface is identical. Motorcycles have a different interface (handlebars), but the same concept applies.

How did you learn this interface? Driver's ed, your parents, documentation. Same idea.

APIs are Conceptual, Not Implementation 40:48

The API defines the **public interface** — what you can do with something. It does not define the **implementation**. `List.add()` implements differently on a `LinkedList` than on an `ArrayList`.

In Java, you enforce APIs through:

1. **Interfaces** — explicitly define the contract
2. **Visibility modifiers** (`public` / `private`) — control what's exposed

But here's a subtlety: the underlying object might have more public members than what the `List` interface defines. So is the "real API" the interface, or all public members of the concrete object? There's no single right answer — it depends on the language and framework. Web APIs, as you'll see, are much less concrete about this than Java interfaces.

> ⚠ **Key Takeaway**
>
> How do you know how to interact with something defined by an API? **Documentation.** If the documentation is wrong, you can't interact with it correctly. This will matter enormously for the group project.

What is a Web API? 44:32

We're not building `List` — we're building **web APIs**. Let's pull that term apart.

Internet vs. World Wide Web 45:53

These are different things:

- **The Internet** — the physical network of interconnected networks. Transport layer: **TCP/IP**.
- **The World Wide Web** — software running on that network. Application layer protocol: **HTTP** (Hypertext Transfer Protocol).

A **web API** is an API that lives on HTTP. The client sends an HTTP request to a server; the server sends an HTTP response back. The flow is always **client initiates, server responds** — the server never initiates a request to the client.

What is a Protocol? 46:22

A protocol is a **set of rules for an interaction**. Classrooms have protocols (instructor talks, students listen, then Q&A). Meeting the King of England has a protocol (walk in, kneel, don't show your back when leaving). Phone calls have a protocol ("Hello?" → greeting → conversation → goodbye).

Humans are good at figuring things out when protocols break. Computers are not. If you right-click the wrong thing, the computer does exactly what you told it — it doesn't infer your intent. Computer protocols must be followed precisely, or communication fails entirely.

HTTP defines the format of messages sent over the network. When you send a message over HTTP, the protocol specifies what that message should look like — both how it's structured when sent and how it's interpreted when received.

Other Application Layer Protocols 52:07

- **FTP** — File Transfer Protocol
- **SMTP** — Simple Mail Transfer Protocol (email)
- **HTTP** — Hypertext Transfer Protocol (the web)

HTTPS 52:40

HTTPS = HTTP + encryption. The actual HTTP message content is identical — the **S** just means the message is encrypted end-to-end (client encrypts → send → server decrypts, and vice versa).

When running locally, you're on plain HTTP (not worth setting up certificates for development). When you deploy the same code to the cloud, HTTPS is handled automatically by the cloud server — your code doesn't change.

Student Question

Q: Can you define your own protocols?

A: We won't cover custom protocols in this course. But you can think of your API definition as a layer on top of HTTP — a kind of custom protocol. Your API says: "send a request to *this* route with *this* data in *this* format." That's a set of rules for interaction, which is a protocol.

Why Web APIs Exist 58:17

The short answer: money.

The Weather Channel has valuable data and compute resources for weather forecasting. Originally, you could only get that through their cable TV channel. With the web, they built a website — more eyes, more ad revenue. But wouldn't it be great if ESPN could show weather for tonight's game? ESPN isn't in the weather business, but weather data brings value to their users (more eyeballs → more ad revenue).

A **web API** lets the Weather Channel expose their data to other code — ESPN's website, a mobile app, any client. The Weather Channel charges for API access. ESPN pays because it drives their own revenue.

Some APIs are free (TMDB, for example). Many are paywalled — especially sports APIs, because gambling money makes them extremely valuable.

Student Question

Q: What's Prisma?

A: Prisma sits between your TypeScript code and the PostgreSQL database. It's an ORM (Object-Relational Mapper) — a framework you interact with that then interacts with the database, so you don't write raw SQL in your TypeScript code. We'll cover it in Weeks 3–4.

Web API Consumers 1:03:42

This is an important distinction:

	Consumer	What They See
Web pages	Humans (via browsers)	Rendered HTML, styled and visual
Web APIs	Code (apps, websites, other APIs)	Structured data (JSON), not meant for human eyes

Show a JSON API response to a non-technical person and they'll say "that looks like computer stuff." That's correct — it's **for code**, not for humans. The **P** in API stands for **Programming**. Just like the Java `List` API is consumed by Java code, a web API is consumed by code — whether that's a React frontend, a mobile app, or another server.

Implementation is Agnostic 1:06:33

In Java, you need Java code to interact with a Java object. Web APIs are different — **both sides are agnostic**:

- The **server** can be Node.js, Python, .NET, Java — anything
- The **client** can be React, Angular, Python, a mobile app — anything

This works because the one thing that's *not* agnostic is **HTTP**. Any language can speak HTTP, so any language can consume or serve a web API. We're implementing ours using **Node.js** (the runtime — think of it as the JVM for server-side JavaScript) with **Express** (the framework for defining routes and handling requests).

Connecting URLs to Java 1:10:11

Mapping web API concepts back to Java to build intuition.

The URL as a Variable 1:11:20

In Java: `names` is a variable that references an object in memory.

In a web API: `http://localhost:3000` is the reference to a specific service (the web API server).

Java	Web API
`names` (variable → object)	`http://localhost:3000` (URL → service)
`names.get(0)` (method call)	`http://localhost:3000/v1/hello` (route/endpoint)
Method parameters	Query strings, route params, request body
Return value	HTTP response (status code + body)

Localhost and Ports 1:12:42

Think of every computer as a **building**:

- The **IP address** (or domain name) is the **building address** — how you find the building
- `localhost` is an alias for `127.0.0.1`, the loopback address — the request never leaves your computer (you're already in the building)
- The **port** is the **room number** — each room can hold one service at a time

There are roughly 65,000 ports available. When you start your server on port 3000, it occupies room 3000. Send a request to port 3001? Empty room — nothing's there. Send it to 3000? Your web API is ready and waiting.

Routes as Method Names 1:18:51

In `http://localhost:3000/v1/input/users`:

- `http://localhost:3000` → the service (like the `names` variable)
- `/v1/input/users` → the **route** that defines the endpoint (like the method name `.get()`)

Who defines the route names? **You, the developer.** The API designer chooses route names just like whoever designed the `List` interface chose method names. The client/consumer doesn't get to decide.

HTTP Responses 1:22:11

In Java, a method can return `void`. In web APIs, **every endpoint must return an HTTP response** — you can never return nothing. The client can ignore the response, but the server always sends one.

Status Codes 1:23:17

Every HTTP response includes a standardized **status code**:

Range	Meaning	Examples
2xx	Success	200 OK, 201 Created
4xx	Client error (you did something wrong)	400 Bad Request, 401 Unauthorized, 403 Forbidden, 404 Not Found
5xx	Server error (something broke on the server)	500 Internal Server Error

Key distinctions:

- **401 vs. 403** — commonly misused. 401 = you're not authenticated (who are you?). 403 = you're authenticated but not authorized (I know who you are, but you can't do this).
- **404** — everyone's seen it on web pages, but in web APIs it's actually less common than other 4xx codes.
- **500 during development** = you broke your code. **500 in production** = something external is actually down (database, third-party API, etc.) — assuming you tested properly.

Lecture Correction — HTTP 418 I'm a Teapot

In lecture, the status code was referenced as "419" or "420" or "428." The actual code is **418 I'm a Teapot** (RFC 2324), added to the HTTP spec as an April Fools' joke. Many web servers return it on April 1st instead of a real error code.

Use Standard Codes

Using the correct HTTP status codes is important for idiomatic API design. If you return a made-up code like 260, clients won't know what to do with it. Stick to the standard codes.

Response Body 1:28:27

The HTTP response has two parts: the **status code** and the **body**. The body can be:

- **Empty** — similar to a `void` method (the response still exists, the body is just empty)
- **HTML** — what you get when you visit a web page in a browser
- **A file** — downloading a resource from the server
- **XML** — older web APIs (20+ years ago)
- **JSON** — modern web APIs (what we'll use exclusively)

JSON (JavaScript Object Notation) responses are always plain text. Keys and values are strings with double quotes:

```
{
  "message": "Hello! You sent a GET request."
}
```

The web API defines what goes in the body — what key-value pairs, what structure. This is part of your API design, and it must be documented.

Sending Data to Web APIs 1:31:03

In Java, `names.get(0)` passes `0` as an argument. Web APIs have **four ways** to send data to an endpoint.

Query Strings 1:31:47

Appended to the URL after a `?`, as key-value pairs separated by `&`:

```
http://localhost:3000/v1/input/search?q=hello&limit=10
```

You've seen these in every Google search URL. Use query strings when the response can be **zero to many results** — searches, filtering, pagination.

Route Parameters 1:34:11

Embedded directly in the URL path:

```
http://localhost:3000/v1/input/users/13
```

The `13` is a route parameter. Use route parameters when you're looking for a **single specific resource** — "give me the user with ID 13." If ID 13 doesn't exist, you get a 404.

Route parameters can appear anywhere in the path (e.g., `/input/13/details/22`), though putting them in non-obvious positions is arguably poor route design.

Request Body 1:36:16

Data sent inside the HTTP request itself (not visible in the URL). Our request bodies will always be JSON. Use the request body for:

- **Large data** — URLs have a size limit on the number of characters
- **Sensitive data** — query strings and route parameters are visible in the URL (browser history, server logs, etc.)

Headers 1:38:00

The fourth way: HTTP headers are packed into the request. This is where **authentication** data goes — API keys, tokens, etc. We'll cover this in detail during the auth weeks.

HTTP Methods and CRUD 1:38:41

This is where the Java analogy breaks down — there's no Java equivalent.

HTTP defines **methods** (also called verbs) that specify the *type* of action you want to perform. You can have the **same route** respond differently depending on the HTTP method:

HTTP Method	**CRUD Operation**	**Purpose**	**Example: `/users`**
POST	**C**reate	Add a new resource	Create a new user
GET	**R**ead	Retrieve a resource	View a user or list users
PUT	**U**pdate (replace)	Replace an entire resource	Replace all user data

HTTP Method	CRUD Operation	Purpose	Example: `/users`
PATCH	**U**pdate (modify)	Modify part of a resource	Update just the user's email
DELETE	**D**elete	Remove a resource	Delete a user

A single endpoint like `/users` can be a POST, GET, DELETE, PATCH, and PUT — five different operations on the same route, distinguished by the HTTP method.

> ⚠ **Follow the Conventions**
>
> Nothing *prevents* you from using DELETE to create resources. But if you do, clients will be confused — "why am I using DELETE to create something?" By convention and good design, line up the five HTTP methods with the four CRUD operations.

Next Time 1:42:52

Code walkthrough moved to next Tuesday. The lecture demo repo is available now — clone it and work through it alongside the guides:

- Intro to Express — deeper than what we'll cover in class
- Routing & Middleware — deeper than what we'll cover in class
- Error Handling & Validation
- API Testing Tools

The check-off is due next Sunday, but treat it like it's due this Sunday — get ahead of the game.

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