lectures # week-2

Node.js Projects & Express Architecture (Tuesday, April 7, 2026)

Lecture Recording

Watch on Panopto

Today's Agenda

1. **Course admin** — site updates, readings, quizzes, guides, check-offs, sprint releases
2. **GitHub Classroom & AI Diary** — troubleshooting access issues
3. **Group project Q&A** — submission workflow, grading, time expectations
4. **TypeScript project structure** — `package.json`, scripts, dependencies, `npm install`, `tsconfig.json`
5. **Express project structure** — config files, `.env`, dependencies vs. dev dependencies
6. **Express code walkthrough** — entry point, middleware pipeline, routes/middleware/controllers architecture

---

Plan & Admin 0:00

Admin updates, then a code-heavy day working through two TypeScript projects — a minimal example and the Express lecture demo.

In Memoriam

The lecture opened with a moment to acknowledge the sudden passing of a fellow CSS classmate the previous week. The loss has been felt by students and faculty alike. If this is affecting you — now or at any point during the quarter — please reach out. The instructor is understanding and will work with you on accommodations.

Campus support resources:

- **PAWS** — confidential counseling and mental health support for students (located in MAT 354)
- **Office of Student Advocacy & Support (OSAS)** — helps students navigate academic concerns, coordinate support, and connect with campus resources during challenging circumstances

---

Course Site Updates 3:24

Readings & Quizzes 3:47

- **Week 2 reading and quiz** are up — What is a Web API? and REST API Design Principles. Much of Thursday's lecture content is covered in these readings.
- **Week 3 reading** (The Relational Model & Data Modeling) is available for those who want to read ahead. If you've taken the database course, this will be review — it's a broad but shallow overview to establish shared vocabulary.
- **Week 4 reading** should be available by end of week.

Guides 5:28

New guides available this week:

- **Async Programming** — Async Concepts and Async in TypeScript. Most of what we do in this class is asynchronous programming, but modern TypeScript with `async` / `await` makes the code look nearly synchronous.
- **Back-End Web API** — all guides are open, covering today's lecture content and beyond
- **Data Access** — SQL Fundamentals, PostgreSQL & Docker Setup, Prisma ORM, and Pagination & Filtering are all available
- **Deployment** — Deploying a Simple Web API is a detailed step-by-step guide for deploying to two cloud services. You'll need this for Sprint 0.

Check-Offs & Sprints 8:08

- **Week 2 check-off** (Running Web API + API Testing) is due Sunday
- **Week 3 check-off** (Proxy API) is available for those working ahead

- **Sprint 0** should be in progress with groups. Not a heavy lift — you're fine if you haven't started yet.
- **Sprint 1** is already released — read ahead to see what's coming

Future sprints will be released on Thursdays, ideally before class so students can come with questions.

GitHub Classroom & AI Diary 15:17

Several students had issues getting into the AI Diary repository through GitHub Classroom. No single reliable fix was identified — some students reported that clicking the link multiple times eventually worked, others found an invitation in GitHub under Organizations.

Troubleshooting tips:

- Check your email for a GitHub invitation
- Go to GitHub → Organizations and look for a pending invitation
- If you used the wrong GitHub account on the first assignment, the instructor can re-link you in GitHub Classroom
- Try the Sprint 0 link separately to help diagnose whether the issue is specific to the AI Diary repository or GitHub Classroom generally

> ⚠ **Action Item**
>
> Everyone should attempt to access the AI Diary by Thursday. If issues persist, let the instructor know for one-on-one troubleshooting.

Group Project Q&A 20:05

Sprint Submission Workflow

Sprint submissions do not go through Canvas. The submission process is:

1. Do all work on **feature branches** (not `main`)
2. When ready to submit, create a PR and **merge into** `main`

3. The instructor reviews the most recent merge into `main` up to the due date

This applies to every sprint — Sprint 0, Sprint 1, and beyond. The only Canvas submissions are the concept reading quizzes.

Check-Off Submission

Individual check-off assignments are peer-verified. When you're done, get on Discord or meet someone in person — share your screen, demonstrate your work, and they give you the check-off.

User Stories and Epics 9:22

Sprint requirements are structured as **user stories**, but many of them are intentionally vague and are really **epics** — large features that should be broken into multiple actionable user stories by the group.

Student Question

Q: What's the difference between an epic and a user story?

A: An **epic** is a big feature or piece of functionality made up of multiple user stories. When you see a user story like "I need to search for movies," don't treat it as a single action item — break it into smaller stories (define the endpoint, handle edge cases, add pagination, etc.). If your group wants to practice using a project board with a backlog, that's great interview prep.

Time Expectations 22:46

For the group project, ideally **5–10 hours per student per week**, though it will vary widely:

- ~1 hour for readings, ~1–2 hours for the quiz and individual check-off, plus project work
- Some weeks will be heavier than others depending on user story distribution
- Groups aiming for the bare minimum MVP with heavy AI assistance might spend very little time; groups that dive deep could spend much more

---

TypeScript Project Structure 28:20

Starting with a minimal TypeScript project to understand the fundamentals before moving to a full Express application. These examples are from the guides.

TypeScript vs. JavaScript 29:01

What makes a file TypeScript rather than JavaScript?

- **Type declarations on variables** — JavaScript doesn't declare types at all. TypeScript puts type annotations after the variable name (unlike Java, which puts them before).
- **Typed function parameters and return types** — not just variables, but function signatures are typed too
- **File extension** — `.ts` instead of `.js` (the extension itself doesn't enforce anything, but tools use it to determine how to process the file)

> **In the Guides**
>
> TypeScript Essentials — Type Annotations covers variable, parameter, and return type syntax in detail.

Compiling vs. Transpiling 31:08

You can't just run a `.ts` file directly (though modern Node.js versions are changing this). Like Java, TypeScript requires a transformation step before execution.

Java comparison:

- Java source → `javac` (compiler) → bytecode (`.class` files) → runs on the **JVM**
- Compilation takes human-readable code and transforms it to something closer to the runtime

TypeScript:

- TypeScript source → `tsc` (TypeScript compiler) → JavaScript (`.js` files) → runs on **Node.js**
- This is technically **transpilation**, not compilation — one language is transformed into another language at the same level of abstraction

Terminology

Academically, TypeScript undergoes **transpilation** (language-to-language transformation), not true compilation (language to lower-level representation). However, the TypeScript community calls `tsc` the "TypeScript compiler," so we'll use "compile" from here on.

Running `tsc index.ts` produces `index.js`. The output looks nearly identical — just with all the type annotations stripped out.

In the Guides

Building & Running TypeScript — Compiling vs. Transpiling and What tsc Actually Produces walk through this process step by step.

`var` vs. `let` / `const` 36:06

The default TypeScript compiler output uses `var`, but in modern JavaScript and TypeScript, **never use `var`**. It has scoping issues that cause common bugs — variables won't be scoped the way you expect coming from Java.

- Use `const` by default — make every variable immutable
- Use `let` only when you genuinely need to reassign the variable
- Think about *why* you need mutation before switching from `const` to `let`

In the Guides

JavaScript for Java Developers — Variables: let and const explains scoping differences and why `var` causes bugs.

`package.json` 37:59

Every Node.js project starts with a `package.json` file. This defines the project — its metadata, scripts, and dependencies. Think of it as the project's configuration manifest.

JSON stands for JavaScript Object Notation. It looks like a JavaScript object with key-value pairs, but it's not one — all keys must be double-quoted strings, and you can't have functions in it. It's a text format for structured data.

Project metadata — name, version, main entry point. Keeping the version number updated is tedious for humans, but AI tools are good at bumping versions as part of your workflow.

> **In the Guides**
>
> Building & Running TypeScript — Understanding package.json covers all the fields, scripts, and dependency sections in detail.

npm Scripts 40:06

Scripts in `package.json` are essentially aliases for terminal commands. Run them with `npm run <script-name>`.

Three key scripts for any TypeScript project:

Script	Command	Purpose
`build`	`tsc`	Compile the entire project (TypeScript → JavaScript)
`start`	`node dist/index.js`	Run from **compiled** code (production mode)
`dev`	`tsx src/index.ts`	Compile in memory and run on the fly (development mode)

Critical difference between `start` and `dev`:

- **`npm start`** runs pre-compiled JavaScript from the `dist/` folder. You must run `npm run build` first — if there's no compiled output, it won't work.
- **`npm run dev`** uses `tsx` to compile only the needed files in memory on the fly — no `.js` files are saved to disk. It watches for changes and recompiles automatically.

> **Common Mistake**
>
> If you're using `npm start` (production mode) and your changes aren't showing up, you probably forgot to run `npm run build` first. When in doubt, use `npm run dev` during development.

Scripts can be any terminal command — they're not limited to project-specific tools.

Dependencies 45:56

Node.js is intentionally minimal — unlike Java's massive standard class library (3,000+ classes including everything in `java.lang`, `java.util`, Swing, and AWT), Node provides just the runtime. Everything else comes in through project-level **dependencies**.

Type	Purpose	Example
dependencies	Required for your code to **run** in production	`express`, `cors`, `dotenv`
devDependencies	Required for **development** only	`typescript`, `eslint`, `jest`

Why is TypeScript a dev dependency? Because by production time, all TypeScript has been compiled to JavaScript — there's no TypeScript in production.

`npm install` and `node_modules` 50:00

Running `npm install` (or `npm i`) reads `package.json`, downloads all dependencies, and places them in the `node_modules` directory.

Even with only 3 declared dependencies, `node_modules` will contain far more packages — each dependency has its own dependencies, which have their own dependencies, and so on. The `package-lock.json` file records the **exact versions** of every package installed (since `package.json` uses version ranges with `^`).

Always `.gitignore` node_modules

`node_modules` is generated and can be enormous. It must be in `.gitignore` — never commit it to version control. Anyone cloning the repo runs `npm install` to regenerate it.

Student Question

Q: What's `package-lock.json`?

A: When you run `npm install`, the `^` (caret) in version strings like `^5.2.0` means "give me the most recent backwards-compatible version." The lock file pins the **exact** version that was actually installed, ensuring consistent builds across machines and team members.

`tsconfig.json` 50:55

Configures the TypeScript compiler — compilation target, output directory, module resolution, etc.

> **In the Guides**
>
> Building & Running TypeScript covers `tsconfig.json` settings, including `outDir`, `target`, `module`, and `strict` mode.

Express Project Structure 57:48

Moving to the full Express lecture demo project. This is the code you'll clone for the Week 2 check-off — all examples this week come from this repo.

> **Lecture Demo**
>
> github.com/UWT-SET-TCSS460-LECTURE-MATERIALS/TCSS-460-Backend-1

Additional npm Scripts 58:57

Beyond `build`, `start`, and `dev`, the Express project includes:

Script	Purpose
`format`	Run Prettier to fix code formatting
`lint`	Run ESLint to check for code quality issues
`test`	Run Jest test suite against the web API

VS Code and WebStorm can be configured to run linting and formatting automatically on save — set this up so you see issues as squiggly underlines rather than running scripts manually.

In the Guides

Testing and documentation are covered on Thursday. Guides: API Testing, API Testing Tools, OpenAPI Documentation.

Key Dependencies 1:01:04

Dependency	Purpose
express	Framework for building the web API — handles routing and request processing
cors	Cross-Origin Resource Sharing — controls which frontends can connect to the API
dotenv	Loads environment variables from a `.env` file
yamljs	Parses YAML files for API documentation (OpenAPI spec)

Lecture Correction — CORS Acronym

In lecture, CORS was expanded as "Cross-Origin Requests Scripting." The correct expansion is **Cross-Origin Resource Sharing**. CORS controls which web origins (domains) are allowed to make requests to your API from a browser.

Student Question

Q: What does adding a dependency version in `package.json` actually do? Is it setting up CORS?

A: No — listing a dependency just tells npm to download that module's code into `node_modules`. It doesn't configure or activate anything. You still need to `import` the module and use it in your code. Don't conflate bringing in a dependency with using it.

Configuration Files 1:04:16

The project root contains several config files:

File	Purpose
`tsconfig.json`	TypeScript compiler settings
`jest.config.ts`	Test framework configuration
`eslint.config.mjs`	Linting rules
`.prettierrc.json`	Code formatting rules (same-line braces, single quotes, etc.)
`open-api.yaml`	API documentation (OpenAPI spec)
`.gitignore`	Files excluded from version control

In the Guides

Building & Running TypeScript — Linting & Formatting covers ESLint and Prettier configuration. OpenAPI Documentation covers the `open-api.yaml` file.

Environment Variables & `.env` 1:06:06

The `.env` file stores **secrets and configuration values** that your code needs but that should never be in source code:

- API keys (e.g., a weather API key you're paying for)
- Database passwords and connection strings
- Port numbers and other environment-specific settings

Access environment variables in code with `process.env.VARIABLE_NAME` — this works identically whether the variable comes from a `.env` file, system environment variables, or cloud platform configuration.

In the Guides

Intro to Express — Using Environment Variables for the Port walks through the `.env` setup and `dotenv` configuration.

`.env` is `.gitignore`d — it should never be committed to version control. VS Code shows ignored files as grayed out in the file explorer.

`.env.example` **is committed** — it lists the required variable keys with empty or default values, so anyone cloning the repo knows what environment variables to set up. Copy `.env.example` to `.env` and fill in your own values.

When deploying to the cloud, you don't send the `.env` file — instead, you configure environment variables through the cloud platform's UI or CLI.

> ⚠ **Don't Skip the .env Setup**
>
> If you clone the demo repo and routes don't work, you probably forgot to create a `.env` file from `.env.example` and add your own API keys.

`.gitignore` 1:15:03

Important entries in `.gitignore`:

- `.env` — secrets must never be committed
- `node_modules/` — dependencies are downloaded via `npm install`, not versioned
- `dist/` — compiled output can be regenerated

Any file that's `.gitignore`d appears **grayed out** in VS Code's file explorer.

Express Code Walkthrough 1:19:17

Walking through the actual TypeScript code, starting from the entry point and tracing the request handling pipeline.

Entry Point: `index.ts` 1:19:24

In Java, every program starts with a `main` method. In Node.js, there's no hard rule — but by strong convention, the entry point is `index.ts` (which compiles to `index.js`). This convention comes from the web, where `index.html` was the default page.

The entry point should be minimal — like a Java GUI class whose `main` method just kicks off the window:

```
import 'dotenv/config';          // Load .env file into process.env
import { app } from './app';     // Import the Express app
```

```
const port = parseInt(process.env.PORT || '3099');

app.listen(port, () => {
    console.log(`Server is running on localhost:${port}`);
});
```

Two imports:

- `dotenv/config` — external module (from `node_modules`), activates `.env` loading
- `./app` — internal module (your own code), indicated by the `./` path prefix

> **In the Guides**
>
> Modules & Imports — Module Resolution explains how Node.js distinguishes relative imports (`./app`) from package imports (`express`).

Port setup: Reads `PORT` from environment variables, falls back to `3099` if not set. The fallback is intentionally different from the `.env` value (e.g., `3000`) — if you see the server running on `3099`, you know your `.env` isn't loading, which alerts you to fix the issue before it causes bigger problems.

`app.listen()` starts the server on the specified port. The callback function runs *after* the server is up and listening — it's not called immediately.

> **Student Question**
>
> Q: How do you stop the server?
>
> A: Press **Ctrl+C** in the terminal. If you accidentally close the terminal without stopping the server, the process keeps running and the port stays occupied. On Mac, use `lsof -i :PORT` to find the process and kill it. If you try to start the server and get a "port already in use" error, this is why.

`app.ts` and the Middleware Pipeline 1:31:01

This is where Express is configured. The `app` object — created by calling `express()` — is the core of the web API.

```
import express from 'express';
import cors from 'cors';
import { routes } from './routes';
import { logger } from './middleware/logger';

const app = express();

// Global middleware — applied to every request
```

```
app.use(cors());                // Allow cross-origin requests
app.use(express.json());        // Parse JSON request bodies
app.use(logger);                // Log incoming requests

// Routes
app.use('/v1', routes);

export { app };
```

Lines with `app.use()` and no route string set up **global middleware** — functions that run on every single HTTP request before any route-specific code.

JavaScript is a **first-class function language** — functions can be stored in variables and passed as parameters to other functions (creating higher-order functions). That's exactly what's happening here: `cors()` returns a function, `express.json()` returns a function, and `logger` is a function — all passed into `app.use()`.

> **In the Guides**
>
> JavaScript for Java Developers — Functions Are Values and Arrow Functions cover first-class functions in detail.

The Middleware Pipeline Concept 1:35:16

When an HTTP request comes in, Express builds a **pipeline of functions** to handle it. Each function in the pipeline can either:

- **Pass the request forward** to the next function (happy path)
- **Send an error response** and stop the pipeline (sad path)

The pipeline has two types of functions:

Type	Role	Where Defined
Middleware	Validation, transformation, logging — all functions *before* the last one	`src/middleware/`
Controller	The final function that does the main work and sends the response	`src/controllers/`

On the happy path, only the controller sends a response. Middleware functions should not send responses on the happy path — they validate, transform, and pass the request along. If

middleware detects a problem (missing query string, invalid input), it sends an error response (e.g., 400) and stops the pipeline.

The benefit: if you build your middleware pipeline correctly, **controllers are mostly on the happy path** — most of the sad-path handling is done before the request ever reaches the controller.

> **In the Guides**
>
> Routing & Middleware — The Middleware Chain covers the pipeline concept and the `next()` function in depth.

Global Middleware Walkthrough 1:42:45

The three global middleware functions from `app.ts`:

1. `cors()` — currently wide open, allowing any browser from any website to connect
2. `express.json()` — parses the HTTP body as JSON. If the body can't be parsed as valid JSON, returns a 400 and stops the pipeline.
3. `logger` — logs request details (timestamp, endpoint, errors) to the console

Directory Structure 1:34:57

The project organizes code into three directories:

```
src/
├── routes/          # Define endpoint names and which functions handle them
├── middleware/       # Validation and transformation functions
└── controllers/     # Final functions that send responses
```

Routes are the wiring — they define the endpoint names and assemble the middleware pipeline for each endpoint. They don't contain business logic.

Middleware contains reusable validation and transformation functions. If multiple routes need the same check, pull it into a shared middleware function.

Controllers contain the final handler functions — database queries, business logic, and sending the HTTP response.

Route Building 1:44:50

Endpoint names are built up hierarchically through nested route files:

```
app.ts:             app.use('/v1', routes)        → /v1
routes/index.ts:    router.use('/hello', hello)   → /v1/hello
                    router.use('/input', input)   → /v1/input
```

Each `index.ts` inside a route folder acts like a **barrel file** — it imports sub-routers and mounts them at path segments. This builds the full endpoint path through composition.

In the Guides

Intro to Express — Project Structure explains the routes/middleware/controllers layout. Routing & Middleware — Organizing Routes in Files covers the barrel file pattern.

HTTP Methods on Routes 1:47:05

At the leaf level, routes bind **HTTP methods** to handler functions:

```
router.get('/', getHello);       // GET    /v1/hello
router.post('/', postHello);     // POST   /v1/hello
router.delete('/', deleteHello); // DELETE /v1/hello
```

The same route path (`/v1/hello`) can respond differently depending on the HTTP method — the same concept from Thursday's lecture about CRUD operations.

Student Question

Q: Why are strings using single quotes instead of double quotes?

A: In JavaScript/TypeScript, single quotes and double quotes are interchangeable for string literals. This project uses single quotes because that's what the Prettier configuration specifies. You can change this in `.prettierrc.json` if your group prefers double quotes.

Next Time

Thursday will cover **API documentation** (OpenAPI) and **API testing** (Jest). We'll continue with this same codebase, looking at the test setup, how to write tests, and how documentation and testing go hand in hand. The guides are already available:

- API Testing
- API Testing Tools

- OpenAPI Documentation

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