guide # tooling

Node.js Setup

TCSS 460 — Client/Server Programming

This guide walks you through installing Node.js, understanding npm, and creating your first TypeScript project from scratch. By the end, you will have a working development environment ready for everything we build this quarter.

---

1 What is Node.js?

If you have written Java, you already know what a runtime is — the JVM (Java Virtual Machine) takes your compiled `.class` files and executes them. Node.js plays the same role, but for JavaScript and TypeScript instead of Java.

Node.js is a JavaScript runtime built on Chrome's V8 engine. It lets you run JavaScript (and, with a little help, TypeScript) outside of a web browser — on your laptop, on a server, or in the cloud. Every web API, every database query, every server-side script we write this quarter runs on Node.js.

1.1 The Java Comparison

Here is how the pieces map from Java to the Node.js world:

Java Ecosystem	Node.js Ecosystem	Purpose
JDK (Java Development Kit)	Node.js	Runtime + development tools
`javac`	`tsc` (TypeScript compiler)	Compiles source code
`java` command	`node` command	Executes compiled/interpreted code
Maven / Gradle	npm	Package manager + build tool

Java Ecosystem	Node.js Ecosystem	Purpose
`pom.xml` / `build.gradle`	`package.json`	Project manifest (dependencies, scripts)
Maven Central	npm Registry	Public package repository
`.java` files	`.ts` files	Source code
`.class` files	`.js` files	Compiled output

The biggest conceptual shift: in Java, you always compile first (`javac`) then run (`java`). In the Node.js world, you can run TypeScript files directly using tools like `ts-node` or even Node.js itself (which now has built-in TypeScript support). We will cover both approaches in this guide.

1.2 Why Node.js for This Course?

TCSS 460 is a full-stack course. Our entire stack — from the database layer to the web API to the front-end — runs on JavaScript/TypeScript. Node.js is the foundation that makes this possible:

- **Back end (Weeks 1-5):** We build Express web APIs that run on Node.js
- **Front end (Weeks 6-10):** We build Next.js applications that also run on Node.js
- **Tooling:** Every tool in our chain (TypeScript compiler, test runners, linters) is a Node.js program

One runtime, one language, one ecosystem — from database to browser.

---

2 Installing Node.js

We will install the current **Long Term Support (LTS)** version of Node.js. LTS versions receive bug fixes and security patches for 30 months, making them the right choice for development work.

Required Version

Install **Node.js 24 LTS** (codename "Krypton"). As of March 2026, the latest LTS release is **v24.14.1**. This version ships with **npm 11**.

Do **not** install Node.js 25 (the "Current" release) — it may include breaking changes that have not been tested with our course tools.

2.1 Option A: Direct Download (Recommended for Most Students)

The simplest approach is to download the official installer from the Node.js website.

1. Go to https://nodejs.org/en/download
2. Select the **LTS** tab (it should be selected by default)
3. Download the installer for your operating system:
   - **macOS:** `.pkg` installer
   - **Windows:** `.msi` installer
   - **Linux:** Follow the instructions for your distribution
4. Run the installer and accept the defaults

The installer includes both `node` (the runtime) and `npm` (the package manager).

Try It Yourself

After installation, open a **new** terminal window (this is important — existing terminals will not pick up the new PATH) and verify both tools are installed:

```
node -v
```

Expected output (your patch version may differ slightly):

```
v24.14.1
```

```
npm -v
```

Expected output:

```
11.9.0
```

If both commands print version numbers, you are ready to go.

2.2 Option B: nvm (Node Version Manager)

If you plan to work with multiple Node.js projects that require different versions, or if you want finer control over your installation, use **nvm**. This is common in professional development environments.

macOS / Linux:

```
# Install nvm (check https://github.com/nvm-sh/nvm for the latest version)
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.40.3/install.sh | bash

# Close and reopen your terminal, then install Node.js 24 LTS
nvm install 24

# Verify
node -v
npm -v
```

Windows:

On Windows, use nvm-windows instead (the original nvm does not support Windows). Download the installer from the releases page, then:

```
nvm install 24
nvm use 24

node -v
npm -v
```

Why nvm?

With nvm, you can switch between Node.js versions instantly:

```
nvm use 24    # Switch to Node.js 24
nvm use 22    # Switch to Node.js 22
nvm ls        # List installed versions
```

You do **not** need nvm for this course — the direct download is fine. But if you continue working with Node.js after TCSS 460, nvm becomes very useful.

2.3 Troubleshooting Installation

🔧 Common Issues

`node: command not found` (macOS/Linux) or `'node' is not recognized` (Windows)

- Close your terminal and open a new one. The installer updates your PATH, but only new terminals pick up the change.
- On macOS, if you use zsh (the default), make sure the Node.js path is in `~/.zshrc`. The installer usually handles this, but nvm requires you to restart the terminal.
- On Windows, try restarting your computer if a new terminal does not work.

Permission errors on macOS/Linux (`EACCES`)

- If you installed Node.js with the direct download and see permission errors when running `npm install -g`, the fix is to change npm's default directory. See the npm docs on resolving EACCES permissions errors.
- Better yet, use nvm — it installs Node.js in your home directory, so you never need `sudo`.

Multiple Node.js installations conflicting

- If you previously installed Node.js through Homebrew, a system package manager, or a different method, you may have conflicting installations. Check with `which node` (macOS/Linux) or `where node` (Windows) to see which `node` binary is being used.
- Pick one installation method and uninstall the others.

Wrong version showing

- Run `which node` (macOS/Linux) or `where node` (Windows) to confirm which binary is running.
- If using nvm: `nvm use 24` to switch to the correct version.

---

3 npm — The Package Manager

When you installed Node.js, you also got **npm** (Node Package Manager). npm does three things:

1. **Installs packages** (libraries and tools) from the npm Registry
2. **Manages dependencies** for your project
3. **Runs scripts** defined in your project

3.1 The Java Comparison

If you have used Maven or Gradle in TCSS 360 or other courses, npm fills the same role:

Maven / Gradle	npm	What it does
`mvn install`	`npm install`	Download all dependencies
`mvn dependency:resolve`	`npm install <package>`	Add a specific dependency
`pom.xml` / `build.gradle`	`package.json`	Declare dependencies and build configuration
`~/.m2/repository`	`node_modules/`	Local cache of downloaded packages
`mvn compile`	`npm run build`	Build/compile the project
`mvn exec:java`	`npm start` or `npm run dev`	Run the project
Maven Central	npmjs.com	Public package registry

The key difference: Maven downloads `.jar` files into a global cache (`~/.m2/`). npm downloads packages into a `node_modules/` folder **inside your project**. This means each project has its own isolated copy of its dependencies — no version conflicts between projects.

3.2 `package.json` — The Project Manifest

Every Node.js project has a `package.json` file at its root. This is the single source of truth for your project: its name, version, dependencies, and scripts.

Here is a minimal example:

```
{
  "name": "my-api",
  "version": "1.0.0",
  "description": "My first Express API",
  "main": "dist/index.js",
  "scripts": {
    "build": "tsc",
    "start": "node dist/index.js",
    "dev": "ts-node-dev --respawn src/index.ts"
  },
  "dependencies": {
    "express": "^5.1.0"
  },
  "devDependencies": {
    "typescript": "^5.8.0",
    "ts-node-dev": "^2.0.0",
    "@types/node": "^24.0.0",
```

```
    "@types/express": "^5.0.0"
  }
}
```

Key sections:

Field	Purpose
`name`	Project name (lowercase, no spaces)
`version`	Semantic version (major.minor.patch)
`scripts`	Named commands you can run with `npm run <name>`
`dependencies`	Packages needed at runtime (Express, Prisma, etc.)
`devDependencies`	Packages needed only during development (TypeScript, linters, test tools)

dependencies vs. devDependencies

Think of it this way: if a package is needed to **run** your application in production, it goes in `dependencies`. If it is only needed to **build, test, or develop** your application, it goes in `devDependencies`.

- `express` — needed at runtime (dependency)
- `typescript` — only needed to compile (devDependency)
- `@types/node` — only needed for type checking (devDependency)

3.3 Essential npm Commands

You will use these commands constantly this quarter:

Initialize a new project:

```
npm init -y
```

Creates a `package.json` with default values. The `-y` flag accepts all defaults (you can edit the file afterward).

Install all dependencies for an existing project:

```
npm install
```

Reads `package.json` and downloads everything listed in `dependencies` and `devDependencies` into the `node_modules/` folder. This is the first command you run after cloning any project.

In Java terms, this is like `mvn install` — it resolves and downloads everything your project needs.

Add a new dependency:

```
npm install express           # Runtime dependency
npm install -D typescript     # Dev dependency (-D = --save-dev)
```

The `-D` flag (short for `--save-dev`) adds the package to `devDependencies` instead of `dependencies`.

Run a script defined in `package.json`:

```
npm run dev       # Runs the "dev" script
npm run build     # Runs the "build" script
npm start         # Shortcut for "npm run start"
npm test          # Shortcut for "npm run test"
```

Run a package binary without installing it globally:

```
npx tsc                    # Run the TypeScript compiler
npx ts-node src/index.ts   # Run a TypeScript file
```

`npx` looks for the command in your local `node_modules/.bin/` first, then falls back to the npm registry. It is the recommended way to run tools installed as devDependencies.

Try It Yourself

Create an empty directory and initialize a project:

```
mkdir hello-node
cd hello-node
npm init -y
```

Open the generated `package.json` in your editor and examine its contents. Then install a package:

```
npm install lodash
```

Look at the changes: `package.json` now lists `lodash` under `dependencies`, and a `node_modules/` directory appeared containing the downloaded code. A `package-lock.json` file was also created — this locks exact dependency versions for reproducible builds.

3.4 Creating Your Own npm Scripts

The `scripts` section of `package.json` is not limited to predefined commands — you can create any script you want. A script is just a name mapped to a shell command. When you run `npm run <name>`, npm executes that command with `node_modules/.bin/` on the PATH, so any locally installed tool is available without `npx`.

Here is an example with several custom scripts:

```
{
  "scripts": {
    "build": "tsc",
    "start": "node dist/index.js",
    "dev": "ts-node-dev --respawn src/index.ts",
    "lint": "eslint src/",
    "lint:fix": "eslint src/ --fix",
    "typecheck": "tsc --noEmit",
    "clean": "rm -rf dist node_modules"
  }
}
```

Script	Command	Purpose
`npm run build`	`tsc`	Compile TypeScript to JavaScript
`npm start`	`node dist/index.js`	Run the compiled app
`npm run dev`	`ts-node-dev --respawn src/index.ts`	Development server with auto-reload
`npm run lint`	`eslint src/`	Check code for style and quality issues
`npm run lint:fix`	`eslint src/ --fix`	Fix auto-fixable lint issues
`npm run typecheck`	`tsc --noEmit`	Type-check without generating output files
`npm run clean`	`rm -rf dist node_modules`	Delete generated files for a fresh start

A few things to notice:

- **`npm start` and `npm test` are special** — they do not need the `run` keyword. Every other custom script requires `npm run <name>`.
- **Colons are just convention**, not syntax. `lint:fix` is a separate script from `lint` — the colon makes it visually clear that it is a variant.

- **Scripts can run anything** — not just Node.js tools. The `clean` script above runs a plain shell command. You can chain commands with `&&` (run the second only if the first succeeds) or call other npm scripts.

> **Try It Yourself**
>
> Add a custom script to your `package.json`:
>
> ```
> "scripts": {
>   "hello": "echo 'Hello from npm!'"
> }
> ```
>
> Run it:
>
> ```
> npm run hello
> ```
>
> You should see `Hello from npm!` in your terminal. Now try something more useful — add a `typecheck` script that runs `tsc --noEmit` and use it to check your project for type errors without generating any output files.

In Java terms, custom npm scripts are like the `<executions>` section of a Maven POM or custom Gradle tasks — named commands that automate common development tasks. The difference is that npm scripts are simpler to write and do not require any special syntax beyond the shell command itself.

3.5 `node_modules/` and `.gitignore`

The `node_modules/` directory can contain thousands of files and hundreds of megabytes. You **never** commit it to Git. Instead, you commit `package.json` and `package-lock.json`, and anyone who clones your repo runs `npm install` to recreate `node_modules/`.

Every Node.js project should have a `.gitignore` that includes:

```
node_modules/
dist/
```

The starter projects for this course already include this, but if you ever create a project from scratch, do not forget it.

> **Never Commit `node_modules/`**
>
> If you accidentally commit `node_modules/`, your repository will become enormous and slow to clone. If this happens, ask for help — removing it from Git history requires special commands.

4 Creating a TypeScript Project from Scratch

Now let us put it all together and create a TypeScript project from an empty directory. This is the process you would follow if no starter project were provided. (For course assignments, you will typically clone a starter repo that already has this set up.)

4.1 Step 1: Initialize the Project

```
mkdir my-ts-project
cd my-ts-project
npm init -y
```

This creates `package.json`. Open it in your editor — you will see the default values.

4.2 Step 2: Install TypeScript and Related Packages

```
npm install -D typescript ts-node @types/node
```

What each package does:

Package	Purpose
`typescript`	The TypeScript compiler (`tsc`) — compiles `.ts` files to `.js`
`ts-node`	Runs TypeScript files directly without a separate compile step
`@types/node`	Type definitions for Node.js built-in APIs (so TypeScript understands `console.log`, `process.env`, etc.)

After running this command, your `package.json` should include:

```
{
  "name": "my-ts-project",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "@types/node": "^24.0.0",
    "ts-node": "^10.9.0",
```

```
    "typescript": "^5.8.0"
  }
}
```

> **Version Numbers**
>
> Your exact version numbers may differ slightly. The `^` prefix means "compatible with" — npm will install the latest minor/patch version within the major version. For example, `^5.8.0` allows `5.8.1`, `5.9.0`, but not `6.0.0`.

4.3 Step 3: Create `tsconfig.json`

The TypeScript compiler needs a configuration file. Generate one with:

```
npx tsc --init
```

This creates a `tsconfig.json` with many commented-out options. For this course, here is a clean starting configuration:

```
{
  "compilerOptions": {
    "target": "ES2023",
    "module": "commonjs",
    "lib": ["ES2023"],
    "outDir": "./dist",
    "rootDir": "./src",
    "strict": true,
    "esModuleInterop": true,
    "skipLibCheck": true,
    "forceConsistentCasingInFileNames": true,
    "resolveJsonModule": true,
    "declaration": true,
    "declarationMap": true,
    "sourceMap": true
  },
  "include": ["src/**/*"],
  "exclude": ["node_modules", "dist"]
}
```

Here is what the key options mean:

Option	Value	Purpose
`target`	`"ES2023"`	Which JavaScript version to compile to
`module`	`"commonjs"`	Module system for the output (Node.js default)

Option	Value	Purpose
`outDir`	`"./dist"`	Where compiled `.js` files go
`rootDir`	`"./src"`	Where your `.ts` source files live
`strict`	`true`	Enable all strict type checking (the whole point of TypeScript)
`esModuleInterop`	`true`	Allows `import express from "express"` syntax
`sourceMap`	`true`	Maps compiled JS back to TS for debugging
`include`	`["src/**/*"]`	Which files to compile
`exclude`	`["node_modules", "dist"]`	Which directories to skip

In Java terms, `tsconfig.json` is like the compiler settings section of your `pom.xml` — it tells the compiler which source version to target, where source files live, and where to put the output.

Use `strict: true`

The `strict` flag enables TypeScript's full type checking. Without it, TypeScript silently allows many of the errors it was designed to catch. Every project in this course uses strict mode. If you see a project without it, that is a red flag.

4.4 Step 4: Create Your First TypeScript File

Create the `src/` directory and a file inside it:

```
mkdir src
```

Create `src/index.ts` with the following content:

```
const greeting: string = "Hello from TypeScript!";
const port: number = 3000;

console.log(greeting);
console.log(`If this were a server, it would run on port ${port}`);

// A simple function with type annotations
function add(a: number, b: number): number {
```

```
  return a + b;
}

console.log(`2 + 3 = ${add(2, 3)}`);
```

Notice the type annotations — `: string`, `: number`, `(a: number, b: number): number`. These are the TypeScript additions that do not exist in plain JavaScript. They tell the compiler (and your editor) exactly what types each variable and parameter should be.

4.5 Step 5: Add npm Scripts

Update the `scripts` section of your `package.json`:

```
{
  "scripts": {
    "build": "tsc",
    "start": "node dist/index.js",
    "dev": "ts-node src/index.ts"
  }
}
```

Script	Command	What it does
`build`	`npm run build`	Compiles all `.ts` files to `.js` in `dist/`
`start`	`npm start`	Runs the compiled JavaScript
`dev`	`npm run dev`	Runs TypeScript directly (no compile step)

Try It Yourself

Run your first TypeScript program:

```
npm run dev
```

You should see:

```
Hello from TypeScript!
If this were a server, it would run on port 3000
2 + 3 = 5
```

Now try the compile-then-run approach:

```
npm run build
npm start
```

Look inside `dist/` — you will find `index.js` (compiled JavaScript) and `index.js.map` (source map). Open `dist/index.js` and compare it to `src/index.ts` — all the type annotations have been stripped away. The compiled JavaScript is what Node.js actually executes.

---

5 Running TypeScript

There are several ways to run TypeScript code. Each has its place, and you will use different approaches depending on the situation.

5.1 Option 1: `ts-node` — Run Directly

`ts-node` compiles and runs TypeScript in a single step, in memory. No `.js` files are created on disk.

```
npx ts-node src/index.ts
```

Or, if you defined a `dev` script in `package.json`:

```
npm run dev
```

When to use: Quick scripts, one-off experiments, and development. This is the fastest way to run TypeScript while you are writing code.

Limitations: Slower startup than running pre-compiled JavaScript because it compiles on every run. Not recommended for production.

5.2 Option 2: `tsc` — Compile First, Then Run

The traditional two-step approach:

```
npx tsc              # Compile: src/*.ts → dist/*.js
node dist/index.js   # Run the compiled JavaScript
```

Or using npm scripts:

```
npm run build   # Compile
npm start       # Run
```

When to use: Production builds and deployment. You compile once, then run the JavaScript as many times as you need. This is what happens when your app is deployed to a cloud server.

5.3 Option 3: Auto-Reload with `ts-node-dev`

During active development, you do not want to manually restart your server every time you change a file. `ts-node-dev` watches your files and automatically restarts when something changes. See Building & Running TypeScript for how `ts-node-dev` fits into the broader compilation pipeline alongside `tsc --watch` and `nodemon`.

Install it:

```
npm install -D ts-node-dev
```

Add a script to `package.json`:

```
{
  "scripts": {
    "dev": "ts-node-dev --respawn src/index.ts"
  }
}
```

Now when you run `npm run dev`, the server starts and automatically restarts whenever you save a file.

Flag	Purpose
`--respawn`	Restart the process on file changes (required for servers)

In Java terms, this is like having `mvn spring-boot:run` with auto-reload — except you do not need Spring Boot or any framework for it to work.

Try It Yourself

1. Install `ts-node-dev`:

```
npm install -D ts-node-dev
```

2. Update your `dev` script in `package.json`:

```
"dev": "ts-node-dev --respawn src/index.ts"
```

3. Run it:

```
npm run dev
```

4. While it is running, open `src/index.ts` in your editor, change the greeting text, and save the file. Watch the terminal — the program restarts automatically with your new output.
5. Press `Ctrl+C` to stop the process.

5.4 Option 4: Node.js Native TypeScript (Experimental)

Starting with Node.js v22.6.0, Node.js can run TypeScript files directly using a feature called **type stripping**. Node.js removes the type annotations and runs the remaining JavaScript — no compiler or extra tool needed.

```
node src/index.ts
```

This works with Node.js 24 without any flags for basic TypeScript files that use only "erasable" syntax (type annotations, interfaces, type aliases). However, some TypeScript features like `enum` and `namespace` require an additional flag (`--experimental-transform-types`).

⚠ Not Recommended for This Course (Yet)

Native TypeScript support in Node.js is still considered experimental and has important limitations:

- It does **not** read your `tsconfig.json` — compiler options are ignored
- It does **not** perform type checking — errors are silently ignored
- Some TypeScript syntax (`enum`, `namespace`) requires extra flags
- Tool compatibility (debuggers, test runners) is still catching up

For this course, use `ts-node` or `ts-node-dev` for development and `tsc` for production builds. These are the battle-tested approaches that our starter projects are configured to use. As native support matures, this recommendation may change.

5.5 Summary of Approaches

Approach	Command	Compiles?	Type Checks?	Auto-Reload?	Use When
`ts-node`	`npx ts-node file.ts`	In memory	Yes	No	Quick scripts, one-off runs
`tsc` + `node`	`npx tsc` then `node dist/file.js`	To `dist/`	Yes	No	Production builds
`ts-node-dev`	`npx ts-node-dev --respawn file.ts`	In memory	Yes	Yes	Active development
`node` (native)	`node file.ts`	Strips types	No	No	Experimental, not yet recommended

---

6 Project Structure Conventions

Every TypeScript project in this course follows the same directory structure. Learning this convention now will make navigating the starter projects much easier.

6.1 Standard Layout

```
my-project/
├── src/                    ← Your TypeScript source code
│   ├── index.ts            ← Entry point
│   ├── routes/             ← Express route handlers
│   ├── middleware/          ← Express middleware
│   └── ...
├── dist/                   ← Compiled JavaScript (generated, never edit)
│   ├── index.js
│   ├── index.js.map
│   └── ...
├── node_modules/           ← Installed dependencies (generated, never commit)
├── package.json            ← Project manifest
├── package-lock.json       ← Locked dependency versions
├── tsconfig.json           ← TypeScript compiler configuration
├── .gitignore              ← Files to exclude from Git
└── README.md               ← Project documentation
```

Directory / File	Purpose	Edit?	Commit to Git?
`src/`	Your TypeScript code	Yes	Yes
`dist/`	Compiled JavaScript output	No (generated)	No
`node_modules/`	Downloaded packages	No (generated)	No
`package.json`	Dependencies and scripts	Yes	Yes
`package-lock.json`	Exact dependency versions	No (auto-updated)	Yes
`tsconfig.json`	Compiler settings	Rarely	Yes
`.gitignore`	Git exclusions	Rarely	Yes

6.2 Complete `package.json` Example

Here is a complete `package.json` for a typical TypeScript project in this course:

```json
{
  "name": "my-express-api",
  "version": "1.0.0",
  "description": "TCSS 460 Express API",
  "main": "dist/index.js",
  "scripts": {
    "build": "tsc",
    "start": "node dist/index.js",
```

```
    "dev": "ts-node-dev --respawn src/index.ts",
    "lint": "eslint src/",
    "test": "jest"
  },
  "dependencies": {
    "express": "^5.1.0"
  },
  "devDependencies": {
    "@types/express": "^5.0.0",
    "@types/node": "^24.0.0",
    "ts-node": "^10.9.0",
    "ts-node-dev": "^2.0.0",
    "typescript": "^5.8.0"
  }
}
```

6.3 Complete `tsconfig.json` Example

And the matching `tsconfig.json`:

```
{
  "compilerOptions": {
    "target": "ES2023",
    "module": "commonjs",
    "lib": ["ES2023"],
    "outDir": "./dist",
    "rootDir": "./src",
    "strict": true,
    "esModuleInterop": true,
    "skipLibCheck": true,
    "forceConsistentCasingInFileNames": true,
    "resolveJsonModule": true,
    "declaration": true,
    "declarationMap": true,
    "sourceMap": true
  },
  "include": ["src/**/*"],
  "exclude": ["node_modules", "dist"]
}
```

6.4 The `.gitignore`

At minimum, your `.gitignore` should contain:

```
# Dependencies
node_modules/

# Compiled output
dist/

# Environment variables (never commit secrets)
.env

# OS files
```

```
.DS_Store
Thumbs.db

# Editor directories
.idea/
.vscode/
```

Try It Yourself

Verify your complete project structure by running:

```
# From your project root
ls -la
```

You should see `package.json`, `tsconfig.json`, `node_modules/`, and your `src/` directory. If you ran `npm run build`, you should also see `dist/`.

Try intentionally introducing a type error in `src/index.ts`:

```
const name: number = "Alice";  // Type error: string assigned to number
```

Run `npm run build` — the TypeScript compiler will report the error and refuse to compile. This is the type safety you get from TypeScript. Fix the error and build again.

Gen AI & Learning: Project Scaffolding

When you start using a coding agent (covered in a separate guide), one of the most common tasks is asking it to scaffold a new project. The agent will generate `package.json`, `tsconfig.json`, and a directory structure for you. Understanding what each file does — which you just learned — is essential for evaluating whether the agent's output is correct. If you do not understand the structure, you cannot verify the scaffold.

---

7 Summary

Concept	Key Point
Node.js	JavaScript/TypeScript runtime — like the JVM but for JS/TS
npm	Package manager — like Maven/Gradle for the Node.js ecosystem

Concept	Key Point
`package.json`	Project manifest — declares dependencies, scripts, and metadata
`node_modules/`	Downloaded packages — generated by `npm install`, never commit to Git
`tsconfig.json`	TypeScript compiler configuration — controls how `.ts` compiles to `.js`
`src/`	Your TypeScript source code lives here
`dist/`	Compiled JavaScript output — generated by `tsc`, never edit directly
`ts-node`	Runs TypeScript directly without a compile step (development)
`ts-node-dev`	Like `ts-node` but auto-restarts on file changes (active development)
`tsc`	TypeScript compiler — compiles `.ts` to `.js` (production builds)
`npx`	Runs package binaries without global installation
`npm run <script>`	Runs a named script from `package.json`
`npm install`	Downloads all dependencies listed in `package.json`
`npm install -D`	Adds a development dependency

---

8 References

Official Documentation:

- Node.js Documentation — API reference and guides for Node.js
- npm Documentation — Complete npm reference including CLI commands and `package.json` specification
- TypeScript Handbook — Official TypeScript language reference
- TypeScript `tsconfig.json` Reference — Every compiler option explained
- Node.js — Running TypeScript Natively — Official guide to Node.js native TypeScript support

Tools:

- ts-node Documentation — TypeScript execution for Node.js
- ts-node-dev on npm — Auto-restarting TypeScript runner
- nvm (Node Version Manager) — Manage multiple Node.js versions

---

9 Further Reading

External Resources

- Node.js Release Schedule — LTS and Current release timelines
- npm `package.json` Documentation — Complete specification for `package.json` fields
- Semantic Versioning (semver.org) — The versioning scheme used by npm packages
- Node.js — Evolving the Release Schedule — How Node.js manages LTS releases

---

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