guide # tooling

Code Quality Tools

TCSS 460 — Client/Server Programming

Your project ships with ESLint and Prettier already configured. You did not install them, you did not write the config files, and you do not need to set anything up. But you *will* see red squiggles in your editor, warnings in your terminal, and failed CI checks if your code does not meet the project's standards. This guide explains what these tools are, why they exist, how they complement the TypeScript compiler, and how to work with them day-to-day.

---

1 Three Layers of Code Quality Checking

Before diving into any specific tool, it helps to understand the big picture. Your TypeScript project has three separate tools checking your code, and each one catches a different category of problems:

Tool	What It Does	What It Catches
`tsc` (TypeScript compiler)	Type-checks and compiles your code	Type errors, missing imports, wrong argument counts, type mismatches
ESLint (static analysis)	Reads your code without running it, looking for patterns that indicate bugs or bad practice	Logic issues, bad patterns, potential bugs that are *valid TypeScript* but still wrong
Prettier (formatter)	Reformats your code to a consistent style	Nothing — it does not find bugs, it just makes code look the same everywhere

These tools have almost zero overlap, and that is by design. Each one owns a distinct layer of quality:

If you have used Java through the 142 and 305 sequence, this layering should feel familiar. `javac` catches type errors, but you still use tools like Checkstyle or SpotBugs to catch style violations and potential bugs that compile just fine. The TypeScript ecosystem splits these responsibilities across three tools instead of bundling them into one.

1.1 What `tsc` Will NOT Catch

The TypeScript compiler is powerful, but it only checks whether your code is *type-safe*. There is a whole category of problems where `tsc` reports zero errors but the code is still wrong. These are exactly the problems ESLint exists to catch.

Unused variables — declared, assigned, never read:

```
function processUser(name: string) {
  const uppercased = name.toUpperCase(); // assigned but never used
  return name;
}
```

`tsc` is happy. ESLint flags `uppercased` as unused.

Using `==` instead of `===` — both compile, one is almost always a bug:

```
function isAdmin(role: string | number) {
  return role == 1; // loose equality — "1" == 1 is true, probably not
intended
}
```

`tsc` sees no type error. ESLint's `eqeqeq` rule catches this and tells you to use `===`.

`let` that should be `const` — a variable that is never reassigned:

```
let port = 3000;
// ... port is never reassigned anywhere
app.listen(port);
```

`tsc` does not care. ESLint's `prefer-const` rule tells you to use `const` because the value never changes.

Floating promises — calling an async function without `await`:

```
async function saveUser(user: User) {
  prisma.user.create({ data: user }); // no await — silently drops errors
}
```

This compiles without error. The function returns before the database write completes, and if the write fails, the error vanishes. ESLint's `@typescript-eslint/no-floating-promises` rule catches this.

`console.log` **left in production code:**

```
app.get("/users", async (request, response) => {
  console.log("DEBUG: hit the users endpoint"); // left from debugging
  const users = await prisma.user.findMany();
  response.json(users);
});
```

`tsc` has no opinion. ESLint's `no-console` rule flags it so you clean up before shipping.

> **Important**
>
> Each of these examples compiles and runs. The TypeScript compiler's job is type safety — it does not enforce code quality, best practices, or consistency. That is why ESLint exists.

---

2 ESLint — Your Automated Code Reviewer

ESLint is a **static analysis** tool. It reads your source code without running it and checks it against a set of rules. Each rule looks for a specific pattern — an unused variable, a missing `await`, a `==` that should be `===` — and reports it as a warning or error.

Think of ESLint as an automated code reviewer that never gets tired and checks the same things every time. A human reviewer might miss that you left a `console.log` on line 47 of a 200-line file. ESLint will not.

2.1 Rules, Plugins, and Config

Every check ESLint performs is a **rule** with a name and a severity:

Severity	Meaning	Effect
`"off"`	Disabled	Rule does not run
`"warn"`	Warning	Shows up in output, does not block anything
`"error"`	Error	Blocks the build and fails CI

Rules live in a **config file**. ESLint uses a flat config format — your project has a file named `eslint.config.js` (or `eslint.config.mjs` / `eslint.config.ts`) in the project root. You do

not need to write this file from scratch — it is already configured for you.

Plugins extend ESLint with rules for specific libraries or languages. Your project uses `@typescript-eslint`, which adds rules that understand TypeScript's type system — things like catching floating promises or flagging `any` types that the base ESLint rules cannot detect.

2.2 Reading ESLint Errors

When ESLint finds a problem, it tells you exactly where and why. Here is what a typical error looks like in the terminal:

```
src/routes/users.ts
  12:7  error  'result' is assigned a value but never used  @typescript-
eslint/no-unused-vars
  25:3  warning  Unexpected console statement               no-console
```

Each line has four parts:

Part	Example	What It Tells You
Location	`12:7`	Line 12, column 7
Severity	`error`	This will block CI
Message	`'result' is assigned a value but never used`	What is wrong
Rule name	`@typescript-eslint/no-unused-vars`	Which rule triggered — your search term

The **rule name** is the most useful part. If you do not understand why a rule exists, search for it — every ESLint rule has a documentation page explaining the rationale and showing examples.

Your editor also shows these inline. In VS Code and WebStorm, ESLint errors appear as red or yellow squiggles under the offending code. Hover over a squiggle to see the rule name and quick-fix options.

2.3 Fixing vs. Suppressing

When ESLint flags something, you have two options:

Fix the code — the rule is right, your code is wrong. This is the correct response the vast majority of the time:

```
// Before: ESLint says 'let' should be 'const'
let port = 3000;

// After: fixed
const port = 3000;
```

Many fixes can be applied automatically. Run:

```
npx eslint --fix src/
```

ESLint will auto-fix straightforward issues like `let` to `const`, removing unused imports, and adding missing `===`.

Suppress the rule — you know better than the rule. This is rare, but sometimes legitimate:

```
// eslint-disable-next-line no-console -- startup message, intentional
console.log(`Server running on port ${port}`);
```

> **Warning**
>
> Never suppress a rule without a comment explaining why. A bare `// eslint-disable-next-line` tells the next person (and your future self) nothing. If you cannot explain why the rule does not apply, the rule is probably right.

> **Try It Yourself**
>
> 1. Open any `.ts` file in your project
> 2. Add a line: `let x = 42;` (unused variable, `let` instead of `const`)
> 3. Save the file — watch for two squiggles: `prefer-const` and `no-unused-vars`
> 4. Hover over each squiggle to read the rule name and message
> 5. Use the quick-fix menu to auto-fix `let` → `const`
> 6. Delete the line when you are done

2.4 Common Rules You Will See

You do not need to memorize every rule. These are the ones that come up most often in course projects:

Rule	What It Catches
`@typescript-eslint/no-unused-vars`	Variables, imports, or parameters that are declared but never used
`eqeqeq`	Using `==` instead of `===` (loose equality is almost always unintended)
`no-console`	`console.log` statements left in production code
`@typescript-eslint/no-floating-promises`	Async function calls without `await` that silently drop errors
`@typescript-eslint/no-explicit-any`	Using `any` as a type — defeats the purpose of TypeScript
`prefer-const`	Using `let` for a variable that is never reassigned

---

3 Prettier — Consistent Formatting Without Arguments

Prettier is an **opinionated code formatter**. It takes your code and reprints it in a consistent style — the same indentation, the same quote style, the same line breaks, every time, for every file, for every person on your team.

Prettier handles:

- Indentation (tabs vs. spaces, indent width)
- Line length (where to wrap long lines)
- Quote style (single vs. double quotes)
- Semicolons (whether to include them)
- Trailing commas
- Bracket spacing

Prettier does **not** check logic, find bugs, or enforce best practices. It has no opinion about whether your code is correct — only about whether it looks consistent. That is ESLint's job.

3.1 Why a Separate Formatter?

Formatting debates — tabs vs. spaces, semicolons vs. no semicolons, single quotes vs. double quotes — are some of the least productive arguments in software development. Every developer has preferences, and none of those preferences affect whether the code works.

Prettier ends the debate by making the choice for you. The team agrees on a `.prettierrc` config file once, and from that point on, everyone's code looks the same. No more noisy diffs full of whitespace changes. No more code review comments about indentation.

If you used IntelliJ IDEA in TCSS 305, you are already familiar with this concept — IntelliJ's auto-formatter does the same thing for Java. Prettier is the TypeScript ecosystem's equivalent.

3.2 Prettier in Your Workflow

Your project already has a `.prettierrc` config file in the project root. You do not need to create or modify it. Here is how you interact with Prettier:

Format on save — the most common workflow. Both VS Code and WebStorm can be configured to run Prettier every time you save a file. If your editor is set up correctly (see the VS Code / WebStorm guide), you will never need to think about formatting — it just happens.

Format from the command line — useful before committing or when CI fails:

```
npx prettier --write src/
```

This reformats every file in `src/` to match the project's Prettier config.

Format before commit — some projects use pre-commit hooks (via tools like `husky` and `lint-staged`) to automatically format files when you run `git commit`. If your project has this configured, formatting happens automatically and you do not need to do anything.

3.3 ESLint + Prettier Together

If ESLint has some formatting-related rules and Prettier has opinions about formatting, won't they conflict? Yes — and that is a solved problem.

The solution is `eslint-config-prettier`, a package that disables every ESLint rule that Prettier handles. Your project already includes this in its ESLint config. The result is a clean separation:

Tool	Responsibility
Prettier	Owns all formatting decisions

Tool	Responsibility
ESLint	Owns everything else (logic, patterns, best practices)

You do not need to configure this yourself. Just know that if you see a formatting issue, Prettier is the tool to fix it (usually by saving the file), and if you see a logic or pattern issue, ESLint is the tool that caught it.

---

4 Working With These Tools Day-to-Day

Now that you know what each tool does, here is what your daily workflow looks like when these tools are in place.

4.1 In Your Editor

Both VS Code and WebStorm show ESLint errors inline — red squiggles for errors, yellow for warnings. When you see a squiggle:

1. **Hover** over it to see the rule name and message
2. **Click the rule name** or search for it to understand why the rule exists
3. **Use the quick-fix** menu (lightbulb icon in VS Code, Alt+Enter in WebStorm) to apply auto-fixes

Prettier reformats your code on save. If a line suddenly rearranges itself when you press Ctrl+S (or Cmd+S), that is Prettier doing its job. You will get used to it quickly.

> **Tip**
>
> If you are not seeing ESLint squiggles or Prettier is not formatting on save, check the VS Code / WebStorm guide to verify your editor extensions are installed and configured.

4.2 From the Command Line

Your project's `package.json` includes scripts for running these tools:

```
# Check for lint errors
npm run lint
```

```
# Auto-fix lint errors where possible
npm run lint:fix

# Format all files
npm run format

# Check formatting without changing files (what CI runs)
npm run format:check
```

Run these before committing, especially if you are not sure your editor is catching everything.

4.3 In CI/CD

Your project's CI pipeline runs ESLint and Prettier checks automatically on every push. If CI fails with a lint or format error, it means your local setup is not catching something that CI caught.

The fix is always the same:

1. Read the CI error output — it tells you the file, line, and rule
2. Fix the issue locally
3. Push again

This is why project-wide consistency matters. In TCSS 460, another team builds a front-end against your API. If your team's code has inconsistent formatting or lint errors in CI, it signals that the codebase is not well-maintained — and the team depending on your API notices.

4.4 Common Mistakes

Suppressing rules without understanding them. If you do not know why a rule exists, look it up before disabling it. The rule is almost always right.

Disabling rules globally. Adding `"no-console": "off"` to the config file turns off the rule for the entire project. Suppress individual lines when needed — do not weaken the rules for everyone.

Ignoring CI failures. A CI failure on lint or format means your local setup missed something. Fix it locally rather than pushing again and hoping it passes.

Committing with lint warnings. Warnings exist because the issue is not severe enough to block you, but they still represent code that should be cleaned up before a pull request.

---

5 Why This Matters Beyond This Course

Every professional TypeScript and JavaScript codebase uses ESLint and Prettier (or equivalent tools). These are not training wheels — they are standard infrastructure.

- **Code review is faster** when formatting is automated. Reviewers focus on logic and design, not indentation and semicolons.
- **Onboarding is easier** when a new developer can run `npm install` and immediately see the same lint rules and formatting as everyone else.
- **Bugs are caught earlier** when static analysis runs on every save and every commit, instead of waiting for someone to notice a floating promise during code review.

In this course, the stakes are concrete: another team builds a front-end against your API. Consistent, clean code is a form of communication. When your code follows the same patterns everywhere, it is easier for both your team and the team depending on you to read, debug, and trust it.

Gen AI & Learning: AI Tools and Code Quality

AI coding assistants generate code that compiles and runs — but they often produce code that triggers lint warnings. Unused variables, inconsistent formatting, `any` types as shortcuts, missing `await` on async calls — these are all common in AI-generated code.

ESLint is your safety net. If an AI tool writes code that violates a project rule, you see it immediately as a red squiggle or a CI failure. Prettier normalizes the formatting so AI-generated code matches the rest of your project.

The key habit: **do not disable lint rules to make AI-generated code pass.** Fix the code instead. The rules exist because the patterns they catch cause real problems — and that is true whether a human or an AI wrote the code.

---

6 Summary

Concept	Key Point
`tsc`	Catches type errors — but valid TypeScript can still be buggy

Concept	Key Point
ESLint	Static analysis that catches logic issues, bad patterns, and potential bugs
Prettier	Opinionated formatter — makes code look consistent, finds no bugs
Three layers	`tsc` + ESLint + Prettier have almost zero overlap, by design
Rule names	Your search term when you do not understand an ESLint error
Fix, don't suppress	Fix the code unless you can explain why the rule does not apply
`eslint-config-prettier`	Disables ESLint formatting rules so Prettier owns all formatting
Already configured	Your project ships with both tools set up — learn to work with them, not install them

---

7 References

Official Documentation:

- ESLint — Getting Started — Introduction to ESLint and configuration
- ESLint — Rules Reference — Complete list of built-in rules with explanations
- typescript-eslint — TypeScript-specific ESLint rules and configuration
- Prettier — What is Prettier? — Overview and philosophy
- Prettier — Integrating with Linters — How eslint-config-prettier works
- eslint-config-prettier — Turns off ESLint rules that conflict with Prettier

---

8 Further Reading

External Resources

- ESLint Configuration Files (Flat Config) — How the `eslint.config.js` format works
- typescript-eslint — Getting Started — Setting up TypeScript-aware linting from scratch
- Prettier vs. Linters — Prettier's own explanation of how it differs from linters

---

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