DEVELOPER TOOLS

Text Editors & IDEs Survey

nano · Sublime Text

IntelliJ IDEA · PyCharm · WebStorm

Choosing the Right Tool for the Job

Capstone: Five Real Tasks, Five Right Tools

Philip Osztromok

Generated with Claude

Table of Contents

Text Editors & IDEs Survey — 9 Chapters

This course exists to cover a real gap: editors and IDEs beyond what's already owned elsewhere on this site. Before covering anything new, it's worth being explicit about exactly what's already covered — and why this course deliberately doesn't repeat it.

What This Course Deliberately Skips

Vim already has its own two-course track (Learning Vim, Vim Intermediate/Advanced) — modal editing, motions, and real depth already live there. **VS Code** already has its own course (VS Code Essentials) covering its extension ecosystem and workflow in depth. **Android Studio** has its own dedicated course too (Android Studio: The IDE Itself) — but for a different reason: that course isn't a general editor survey at all, it's specifically about Android development tooling, which happens to be built on a JetBrains foundation this course also touches, from a completely different angle.

What This Course Actually Covers

Three genuinely uncovered areas: **nano**, a minimal, always-available terminal editor; **Sublime Text**, a lightweight GUI editor sitting between a minimal tool and a full IDE; and the **JetBrains family** — IntelliJ IDEA, PyCharm, and WebStorm — sharing the same underlying platform Android Studio: The IDE Itself's own Chapter 1 already introduced, here covered from a general, non-Android angle.

A Spectrum, Not a Random List

These aren't an arbitrary grab-bag — they sit along one genuine spectrum of weight and capability: **nano** (minimal, instant startup, no plugin ecosystem) → **Sublime Text** (lightweight, fast, a real but modest plugin ecosystem) → the **JetBrains family** (full IDE, deep language-aware tooling, real startup and memory cost). Vim and VS Code sit at their own points along this identical spectrum too — this course simply doesn't re-teach them, since `vim1`/`vim2` and `code1` already do.

TOOL	WHERE IT SITS	OWNED BY
Vim	Minimal, modal, terminal-based	Learning Vim / Vim Intermediate/Advanced
nano	Minimal, non-modal, terminal-based	This course
Sublime Text	Lightweight GUI	This course
VS Code	Lightweight-to-mid GUI, huge extension ecosystem	VS Code Essentials
JetBrains family	Full IDE	This course (general) / Android Studio (Android-specific)

BUILDING ON ANDROID STUDIO'S OWN FOUNDATION

Android Studio: The IDE Itself's own Chapter 1 already explained that Android Studio is built on IntelliJ IDEA Community Edition, and drew the line between features inherited from that shared platform versus Android-specific additions. This course's own JetBrains chapters (4 onward) build directly on that same architectural understanding rather than re-deriving it — readers who've taken that course already have the foundation this course needs.

MORE FEATURES ISN'T AUTOMATICALLY "BETTER"

It's tempting to treat this spectrum as a strict improvement ladder — nano worse, Sublime Text better, a full IDE best. A full IDE's own real memory usage, startup time, and general complexity are genuine costs, not a free bonus layered on top of a minimal editor's own capability. Which tool is actually best depends entirely on the task at hand — a theme this course returns to directly in Chapter 8's own decision framework.

Hands-On Exercises

EXERCISE 1

A friend asks why this course doesn't cover Vim, since it's clearly a text editor. Explain the actual reason, using this chapter's own material.

View solution

EXERCISE 2

Explain why Android Studio: The IDE Itself isn't considered part of this course's own "JetBrains family" coverage, even though Android Studio is itself built on the IntelliJ Platform.

View solution

EXERCISE 3

A developer claims a full JetBrains IDE is "just strictly better" than nano in every situation, since it has vastly more features. Using this chapter's own warning box, explain what's wrong with this claim.

View solution

CHAPTER 1 QUICK REFERENCE

- This course **skips Vim and VS Code** (already owned by their own courses) and **Android Studio** (Android-specific, not a general editor survey)
- This course covers **nano, Sublime Text, and the JetBrains family** (IntelliJ IDEA, PyCharm, WebStorm)
- These sit on a genuine **spectrum** — minimal → lightweight → full IDE — not an arbitrary list
- The JetBrains chapters build directly on **Android Studio: The IDE Itself's own Chapter 1** rather than re-deriving the shared platform
- More features is **not automatically better** — weight and complexity are real costs, matched against actual task needs in Chapter 8

Chapter 1 placed nano at the minimal end of this course's own spectrum. This chapter covers why it's built that way on purpose — a design philosophy, not a lack of ambition.

nano's Own Design Philosophy

Unlike Vim's own modal editing, nano is **non-modal** — whatever you type is inserted immediately, with no separate insert/normal mode distinction to learn first. Its most visible design decision is the row of on-screen keybinding hints permanently shown at the bottom of the screen (`^X Exit`, `^O Write Out`, and so on) — explicitly designed so a first-time user can be productive immediately, without memorizing anything in advance.

Essential Keybindings

```
^O  Write Out (save)
^X  Exit
^K  Cut line
^U  Uncut (paste)
^W  Where is (search)
^_  Go to line number
```

The `^` caret shown in nano's own on-screen hints means **Ctrl** — `^O` is `Ctrl+O`. This same on-screen legend is genuinely nano's own primary interface convention: the hints aren't a beginner tutorial that disappears later, they're a permanent, always-visible part of the editor.

What nano Deliberately Doesn't Have

No plugin ecosystem, no built-in language server or code completion, no project or workspace concept at all. Editing exactly one file at a time is the assumption baked directly into nano's own design — this is a deliberate scope boundary, not a missing feature waiting to be added.

When nano Is Genuinely the Right Choice

Editing a single configuration file during a remote SSH session; a quick, one-off edit where even Vim's own modal learning curve (for someone who hasn't already invested in it) would cost more time than it saves; or simply the practical reality that nano is already installed and available with zero setup on many minimal server installs and containers, where nothing heavier may even be present at all.

ASPECT	NANO	A FULL IDE
Startup time	Instant	Real, sometimes noticeable
Learning curve	Minimal — hints always visible	Real investment required
Editing model	One file at a time, non-modal	Full project/workspace awareness
Best use case	A quick, single-file remote edit	Sustained, multi-file project work

EXACTLY THE TASK SETTING UP A WEB SERVER ON DEBIAN RELIES ON

Editing an Apache configuration file directly on a remote server over SSH — a real, recurring task in Setting Up a Web Server on Debian's own material — is precisely the kind of job nano (or Vim) is suited for, not a full IDE that isn't even installed on that server and wouldn't add anything for a single-file edit anyway.

REACHING FOR NANO ISN'T A SIGN OF LACKING SKILL

It's easy to assume nano is "only for beginners who don't know a real editor." The more accurate framing, per Chapter 1's own spectrum, is the opposite: an experienced developer confidently reaching for nano for a 30-second remote edit — rather than firing up a heavier tool purely out of habit — is matching tool weight to task exactly as intended, not settling for something lesser.

Hands-On Exercises

EXERCISE 1

You're connected to a remote server over SSH and need to quickly fix one line in a config file. List the nano keybindings, in order, you'd use to open the file, find the line, make the edit, save, and exit.

View solution

EXERCISE 2

A colleague says "nano doesn't even have a plugin system or code completion, so it's clearly an inferior editor." Using this chapter's own material, explain why this misses the point of what nano is actually designed to do.

View solution

EXERCISE 3

A senior developer opens nano to make a one-line fix to a config file on a production server, rather than opening their usual full IDE. Using this chapter's own warning box, explain why this is not a sign of reduced skill or laziness.

View solution

CHAPTER 2 QUICK REFERENCE

- nano is **non-modal** — text inserts immediately, no separate insert/normal mode like Vim
- On-screen keybinding hints (`^X`, `^O`, etc. — `^` means Ctrl) are a **permanent part of the interface**, not a disappearing tutorial
- No plugins, no code completion, no project concept — **a deliberate scope boundary**, not a missing feature
- Genuinely the right choice for a **quick, single-file remote edit** — not just a fallback for beginners

Chapters 1 and 2 placed Sublime Text between nano's own minimalism and a full IDE's own depth. This chapter covers what actually fills that middle ground: a real graphical editor with genuine power features, still fast enough to feel closer to nano than to a full IDE in daily use.

Fast Startup, Real GUI

Unlike nano, Sublime Text is a genuine graphical application — syntax highlighting, a sidebar file browser, tabs across open files. Unlike a full IDE, it's built around launching near-instantly, a deliberate design priority rather than an accident of being simpler.

Multiple Cursors

Sublime Text's own signature feature: `Ctrl+Click` to place additional cursors anywhere, or `Ctrl+D` to select the next occurrence of the currently selected text and add a cursor there too. Every cursor edits simultaneously, letting you retype several similar lines at once, live, without a macro or a find-and-replace pattern. This is genuinely different from Android Studio: The IDE Itself's own **Rename** refactoring — multiple cursors is a manual, visual, multi-point edit you control interactively, not a structural operation that understands what an identifier actually refers to.

Goto Anything

```
Ctrl+P           fuzzy-match jump to any file
Ctrl+P @symbol   jump to a symbol within a file
Ctrl+P :42       jump to line 42
```

Goto Anything (`Ctrl+P`) fuzzy-matches file names for fast, keyboard-only navigation, with `@` narrowing to symbols and `:` jumping to a specific line number — quick project navigation without Sublime Text needing to build the kind of deep, persistent project index a full IDE maintains.

Package Control: A Real But Lighter Plugin Ecosystem

Sublime Text has a genuine package manager, **Package Control** — a real ecosystem nano has none of at all. But it's lighter than a full IDE's own built-in tooling in a meaningful way: packages here are community-contributed and optional, adding capability on top of a fast core editor, rather than being deeply integrated, always-on infrastructure the way a full IDE's own built-in refactoring and language-server tooling is.

ASPECT	NANO	SUBLIME TEXT	FULL IDE
Interface	Terminal only	Real GUI	Real GUI
Startup time	Instant	Near-instant	Real, noticeable
Plugin ecosystem	None	Package Control, optional	Deep, built-in tooling
Project awareness	None	Light (Goto Anything)	Full project index

A GENUINELY POPULAR CHOICE FOR STRAIGHTFORWARD WEB WORK

Sublime Text is a real, common choice for HTML/CSS/JavaScript work covered in this site's own web development courses — precisely because it's fast and capable enough for that kind of project without a full IDE's own overhead, when deep language-server tooling isn't the deciding factor.

MULTIPLE CURSORS ISN'T JUST FIND-AND-REPLACE WITH EXTRA STEPS

It's tempting to treat multiple cursors as basically the same thing as find-and-replace with a regex pattern. They solve genuinely different problems: find-and-replace applies one fixed transformation blindly to every match at once, with no per-location judgment involved. Multiple cursors let you see and interactively edit each location as you go — adjusting one location slightly differently, or simply skipping one that doesn't actually fit the pattern — a level of live, per-instance control find-and-replace doesn't offer at all.

Hands-On Exercises

EXERCISE 1

You need to rename a variable used on five lines, but on two of those lines the surrounding code also needs a small additional tweak specific to that line. Explain why multiple cursors would handle this situation better than a single find-and-replace operation.

View solution

EXERCISE 2

Explain why Sublime Text's own multiple cursors and Android Studio's own Rename refactoring, despite both being described as ways to change several things across code, are not interchangeable tools for the same job.

View solution

EXERCISE 3

A developer working on a straightforward static HTML/CSS/JS website is deciding between a full JetBrains IDE and Sublime Text. Using this chapter's own material, explain a reasonable case for choosing Sublime Text here.

View solution

CHAPTER 3 QUICK REFERENCE

- Sublime Text is a real GUI editor with **near-instant startup**, sitting between nano and a full IDE
- **Multiple cursors** (`Ctrl+Click` / `Ctrl+D`) — interactive, per-location editing, genuinely different from a blind find-and-replace
- **Goto Anything** (`Ctrl+P`, `@`, `:`) — fast fuzzy navigation without a deep project index
- **Package Control** — a real but optional, lighter-weight plugin ecosystem than a full IDE's own built-in tooling

Android Studio: The IDE Itself's own Chapter 1 already explained the shared IntelliJ Platform and drew the line between inherited and Android-specific features. This chapter applies that exact same framework across the whole JetBrains family — not just Android Studio — as the bridge into the language-specific chapters that follow.

Recap: The Shared IntelliJ Platform

JetBrains builds several distinct IDEs from one shared core platform, each a different "flavor" with a specific set of language plugins bundled in. Android Studio: The IDE Itself's own Chapter 1 covers this lineage in full depth — this chapter doesn't re-derive it, just applies it more broadly.

The Family Members This Course Covers

IntelliJ IDEA is the flagship, Java/Kotlin-first. **PyCharm** is Python-focused. **WebStorm** is JavaScript/TypeScript-focused. Other family members exist too — Rider for C#, CLion for C/C++, RubyMine for Ruby — not covered individually here, since each one repeats the identical pattern: the same shared platform, plus a different language's own plugin set bundled in. Naming them is enough to show the pattern generalizes well beyond the three this course covers directly.

What's Shared vs. What's Language-Specific

Shared across every family member, identical to what Android Studio's own Chapter 1 already established: the general editor experience, refactoring tools (Rename, Extract Method), Git integration, and the plugin system itself. Language-specific: each IDE's own code completion and analysis engine for its particular language, framework-aware tooling (Django support in PyCharm, React/Vue awareness in WebStorm, the Android SDK in Android Studio), and which plugins ship bundled by default.

Ultimate vs. Community/Free Editions

JetBrains uses a two-tier model worth knowing before picking a tool: **IntelliJ IDEA** and **PyCharm** each have a free **Community** edition alongside a paid **Ultimate** edition with additional framework support and tooling. **WebStorm** is commercial-only, with a free trial but no free tier. **Android Studio** itself is entirely free, built on IntelliJ IDEA Community — a genuinely practical, real-cost detail, not just a technical footnote.

IDE	PRIMARY FOCUS	FREE TIER?
IntelliJ IDEA	Java, Kotlin	Yes (Community)
PyCharm	Python	Yes (Community)
WebStorm	JavaScript, TypeScript	No (trial only)
Android Studio	Android development	Yes (entirely free)

THE SAME FRAMEWORK, APPLIED MORE BROADLY

This chapter is genuinely just Android Studio Chapter 1's own inherited-vs-specific framework, applied across the entire JetBrains family rather than one member of it. Recognizing that framework once means it applies immediately to any new JetBrains IDE encountered later, not just the three covered in this course.

SHARED PLATFORM DOESN'T MEAN EVERY FEATURE EXISTS EVERYWHERE

It's easy to assume skills transfer 100% identically across every JetBrains IDE just because they share a platform. The general editor experience and refactoring genuinely do transfer directly — but a language-specific feature learned in PyCharm, like one of its own Python-specific debugger extensions, may have no direct equivalent in WebStorm at all, since that particular tooling was built specifically for Python's own ecosystem, not JavaScript's. "Shared platform" means the general experience transfers; it doesn't mean every specific feature exists in every family member.

Hands-On Exercises

EXERCISE 1

A developer switching from PyCharm to WebStorm for a new project expects the Rename refactoring to work identically. Explain whether this expectation is reasonable, using this chapter's own material.

View solution

EXERCISE 2

The same developer also expects a Python-specific debugging feature they relied on heavily in PyCharm to be available in WebStorm. Explain why this expectation is NOT reasonable, using this chapter's own warning box.

View solution

EXERCISE 3

A student on a tight budget wants to learn Java using IntelliJ IDEA, and separately wants to learn JavaScript using WebStorm. Using this chapter's own material on editions, explain the real cost difference between these two choices.

View solution

CHAPTER 4 QUICK REFERENCE

- This chapter applies Android Studio's own **inherited-vs-language-specific framework** across the whole JetBrains family, not just Android Studio
- **IntelliJ IDEA** (Java/Kotlin), **PyCharm** (Python), **WebStorm** (JS/TS) — plus Rider, CLion, RubyMine following the same pattern
- Shared: general editor experience, refactoring, Git integration. Language-specific: completion engine, framework tooling, bundled plugins
- **IntelliJ IDEA**/**PyCharm** have free Community editions; **WebStorm** is commercial-only; **Android Studio** is entirely free
- Shared platform means the **general experience** transfers — it does **not** mean every specific feature exists in every family member

Chapter 4 introduced IntelliJ IDEA as "the flagship" in passing. This chapter explains what that word actually means — it isn't just marketing language, it reflects IntelliJ IDEA's genuine, literal position as the ancestor of every other IDE this course has discussed.

Why "Flagship"

IntelliJ IDEA is the original product the entire JetBrains Platform was built around. PyCharm, WebStorm, Android Studio, and every other family member is **derived from IntelliJ IDEA's own codebase**, not the reverse — IntelliJ IDEA genuinely is the ancestor, not merely one equal member among several.

Java & Kotlin-First Tooling

IntelliJ IDEA's code analysis is deepest specifically for Java and Kotlin — the languages it was originally built around and remains most tuned for. **Kotlin** itself is a JetBrains-created language, so IntelliJ IDEA's own Kotlin support is naturally first-class, often ahead of Kotlin tooling available anywhere else, since the same company builds both the language and its primary IDE.

The Full Plugin Ecosystem

Because IntelliJ IDEA *is* the base platform every other family member derives from, its own plugin marketplace is the largest and most complete across the whole family — including plugins adding support for languages and frameworks the specialized IDEs don't otherwise include at all. A genuinely practical use case: a developer doing primarily Java work who occasionally needs light Python or JavaScript support might simply install a plugin inside IntelliJ IDEA, rather than switching to PyCharm or WebStorm entirely just for occasional, secondary work.

Community vs. Ultimate

Chapter 4 already flagged this distinction; here's the real substance behind it. **Community** is free and open-source, covering Java, Kotlin, Groovy, Scala, and core JVM tooling well. **Ultimate** is paid, adding web framework support, enterprise frameworks like Spring, database tools, and additional languages via bundled plugins not available in Community at all — a genuine, meaningful difference for professional backend and enterprise work, not a nag screen nudging toward an unnecessary upgrade.

ASPECT	COMMUNITY (FREE)	ULTIMATE (PAID)
Core JVM languages	Java, Kotlin, Groovy, Scala	Same, plus deeper enterprise tooling
Web frameworks	Not included	Spring, and others
Database tools	Not included	Included
Other languages	Via community plugins only	Additional bundled plugins

THE NATURAL TOOL FOR THIS SITE'S OWN JAVA AND KOTLIN COURSES

Java Fundamentals/Intermediate-Advanced and Kotlin Fundamentals/Intermediate are both naturally worked through in IntelliJ IDEA — the exact IDE those languages' own tooling was built around first, per this chapter's own material.

"THE ANCESTOR PLATFORM" DOESN'T MEAN "THE BEST PICK FOR EVERY LANGUAGE"

It's tempting to assume that because IntelliJ IDEA is the flagship and the literal ancestor of PyCharm and WebStorm, it must be strictly the best choice for every language, including Python or JavaScript specifically. In practice, PyCharm and WebStorm genuinely bundle deeper, more polished language-specific tooling out of the box for their own focus language than IntelliJ IDEA plus a plugin typically provides. Being the ancestor platform means being the best pick for the languages it was actually built around — Java and Kotlin — and a flexible, capable option via plugins for everything else, not an automatic best choice regardless of language.

Hands-On Exercises

EXERCISE 1

Explain, in your own words, what makes IntelliJ IDEA genuinely "the flagship" rather than just one equally-ranked member of the JetBrains family.

View solution

EXERCISE 2

A team working primarily in Kotlin, with one developer occasionally needing to edit a small Python script, is deciding whether that developer should install a Python plugin in IntelliJ IDEA or switch to PyCharm for that occasional work. Using this chapter's own material, explain a reasonable recommendation.

View solution

EXERCISE 3

A developer argues that since IntelliJ IDEA is the ancestor of PyCharm, using IntelliJ IDEA with a Python plugin must be at least as good as using PyCharm itself for serious daily Python development. Using this chapter's own warning box, explain why this argument doesn't hold up.

View solution

CHAPTER 5 QUICK REFERENCE

- IntelliJ IDEA is the **literal ancestor** platform every other JetBrains IDE is derived from — not just an equally-ranked family member
- Deepest tooling for **Java and Kotlin** specifically — Kotlin being JetBrains' own language
- The **largest plugin ecosystem** in the family, since it is the base platform others derive from
- **Community** (free): Java/Kotlin/Groovy/Scala core tooling. **Ultimate** (paid): web frameworks, Spring, database tools, more languages
- Being the ancestor means **best for Java/Kotlin** — not automatically the best choice for every other language over its own dedicated IDE

Chapter 5 closed by saying PyCharm's own Python-specific tooling genuinely runs deeper than IntelliJ IDEA plus a plugin. This chapter is exactly that tooling, built on top of the shared platform introduced in Chapter 4.

Virtual Environment Management

PyCharm provides a built-in UI for creating and selecting a Python virtual environment per project — venv and conda are both supported directly — and it automatically detects imports in code that reference a package not yet installed, offering to install it right there. A genuine convenience over manually activating a venv and running `pip install` from a separate terminal every time.

Django-Aware Tooling

PyCharm offers Django-specific project templates, syntax highlighting and completion for Django Template Language directly inside `.html` files, and a dedicated structure view for a Django project's own models, views, and URLs. This is a concrete, specific example of Chapter 5's own abstract Community/Ultimate distinction — Django support is an **Ultimate-only** feature, not available in the free Community edition at all.

Scientific Computing Support

PyCharm integrates Jupyter notebooks directly inside the IDE, and its own DataFrame viewer shows a pandas or NumPy structure as an actual interactive table — sortable, scrollable, genuinely explorable — rather than a truncated printed `repr()` string in a console. A dedicated scientific mode layout arranges the IDE around this kind of exploratory data work specifically.

The Integrated Python Debugger

The debugging UI itself — breakpoints, stepping, the Variables pane — is largely inherited from the shared platform, per Chapter 4's own framework. PyCharm's own Python-specific addition on top: viewing a pandas DataFrame's actual contents directly inside the debugger's own variable inspector while paused, rather than only a truncated `repr()` string — the same scientific-computing convenience from above, now available mid-debug.

FEATURE	COMMUNITY	ULTIMATE
venv/conda management	Included	Included
Django-aware tooling	Not included	Included
Jupyter/DataFrame viewer	Not included	Included
Core Python editing/debugging	Included	Included

DIRECTLY RELEVANT TO THIS SITE'S OWN DATA SCIENCE & ML COURSES

Data Science Fundamentals and Machine Learning Fundamentals both work heavily with pandas and NumPy — PyCharm's own scientific mode and DataFrame viewer are genuinely useful tools for that specific coursework, not a generic feature unrelated to it.

THE GUI DOESN'T REPLACE UNDERSTANDING VENV/PIP/CONDA THEMSELVES

Echoing Android Studio: The IDE Itself's own point about Git integration, PyCharm's own virtual environment UI is a convenience layer over the same underlying tools — it doesn't replace needing to understand what a virtual environment actually is and why it matters. The real risk: not knowing how to reproduce the same environment outside PyCharm entirely, for instance on a deployment server that has no PyCharm installed at all and only understands `venv` / `pip` / `conda` directly from the command line.

Hands-On Exercises

EXERCISE 1

A developer using PyCharm's free Community edition wants Django-specific project templates and structure views. Explain whether this is possible, and why, using this chapter's own material.

View solution

EXERCISE 2

A student has only ever created Python virtual environments by clicking through PyCharm's own UI, and has never manually run venv or pip from a terminal. They now need to set up their project's environment on a production server with no PyCharm installed. Using this chapter's own warning box, explain the risk they're facing.

View solution

EXERCISE 3

Explain why PyCharm's own DataFrame viewer, available in both the editor and the debugger, is a genuinely different experience from printing a pandas DataFrame to the console with print(df).

View solution

CHAPTER 6 QUICK REFERENCE

- **Virtual environment management** — built-in venv/conda UI, auto-detecting missing packages from imports
- **Django-aware tooling** — templates, DTL support, structure view — an **Ultimate-only** feature
- **Scientific computing support** — Jupyter integration, an interactive DataFrame viewer in both the editor and the debugger
- The debugging UI itself is largely **inherited**; PyCharm's own Python-specific addition is the DataFrame viewer inside it
- The venv UI is a **convenience layer**, not a substitute for understanding venv/pip/conda directly — needed for any environment outside PyCharm itself

Chapter 6 applied the shared-base-plus-language-specific-depth lens to PyCharm. This chapter applies the identical lens to WebStorm — the same inherited debugging UI, pointed at Node.js instead of the JVM or Android, plus genuinely framework-aware tooling on top.

Node.js Debugging Built In

WebStorm can attach directly to a running Node.js process, or launch and debug an npm script straight from the IDE, without manually configuring a separate debugger-attach flow first. The breakpoints, stepping, and variable inspection are the same inherited debugging UI this course has already traced back to the shared platform — here simply pointed at a Node.js process instead of a JVM or Android runtime.

Framework-Aware Completion (React/Vue/Angular)

WebStorm understands JSX, Vue's own single-file-component syntax, and Angular templates specifically — not generic JS/TS completion applied blindly to files that happen to have a different extension. Completion works inside a `.vue` file's own template block; JSX completion is aware of a React component's actual prop types; Angular's own dependency-injection model lets WebStorm navigate directly between a component and its template. This is genuinely framework-specific tooling, not a generic language server stretched to cover unfamiliar syntax.

TypeScript-First Tooling

WebStorm was built with TypeScript as a first-class citizen from early on, not bolted on afterward — type-aware refactoring, reliable jump-to-definition across `.ts`/`.tsx` files, and inline type errors shown live as you type all reflect that deep, original integration.

No Free Community Edition

Revisiting Chapter 4's own point concretely: unlike IntelliJ IDEA and PyCharm, WebStorm has **no free Community tier at all** — only a time-limited free trial. This is a real, practical factor worth weighing directly against VS Code Essentials' own coverage of VS Code, which offers comparable JS/TS tooling for free via extensions.

FEATURE	WEBSTORM	VS CODE
Cost	Commercial, trial only	Free
JS/TS support	Built in, deeply integrated	Via extensions, genuinely strong
Framework-aware tooling	Built in out of the box	Requires selecting the right extensions
Node.js debugging	Built in	Built in

A NATURAL FIT FOR THIS SITE'S OWN JS/TS/FRAMEWORK COURSES

JavaScript Fundamentals/Intermediate/Advanced, TypeScript's own four-course track, React, Vue, and Angular all pair naturally with WebStorm's own framework-aware tooling — genuinely useful, not incidental, for working through that coursework.

"NO FREE TIER" DOESN'T AUTOMATICALLY MEAN "VS CODE IS STRICTLY BETTER"

It's tempting to conclude that since WebStorm has no free tier, VS Code must simply be the better choice for JS/TS work. VS Code's own JS/TS support is genuinely excellent via extensions — much of it built on the same underlying TypeScript language service WebStorm itself uses. WebStorm's own refactoring tools and framework-aware navigation are generally more deeply integrated and reliable out of the box, without needing to find and configure the right extensions yourself. Price isn't the only axis — the real trade-off is "free and very good" versus "paid and somewhat more integrated by default," not "free is simply worse."

Hands-On Exercises

EXERCISE 1

Explain why WebStorm's ability to debug a Node.js script is described as using "the same inherited debugging UI" as other JetBrains IDEs, rather than something built specifically and separately for Node.js.

View solution

EXERCISE 2

A student on a tight budget concludes that because WebStorm costs money and VS Code is free, VS Code must simply be the objectively better choice for React development. Using this chapter's own warning box, explain what's missing from this reasoning.

View solution

EXERCISE 3

Explain what "framework-aware completion" specifically means for a .vue file in WebStorm, and why this is different from just applying generic JavaScript completion to a file with a .vue extension.

View solution

CHAPTER 7 QUICK REFERENCE

- **Node.js debugging** uses the same inherited breakpoint/stepping/variable UI, pointed at a Node process instead of the JVM/Android
- **Framework-aware completion** for React/Vue/Angular — genuinely template/JSX/component-aware, not generic JS completion
- TypeScript was a **first-class citizen** in WebStorm from early on, not added afterward
- WebStorm has **no free Community tier** — only a trial, unlike IntelliJ IDEA/PyCharm
- The real WebStorm-vs-VS-Code trade-off is **"paid and more integrated" vs. "free and very good"** — not a simple free-is-worse comparison

Chapter 8 · Choosing Between a Minimal Editor, a Lightweight Editor, and a Full IDE

Chapter 1 introduced a spectrum; every chapter since has filled in one point on it. This chapter turns that spectrum into an actual decision framework — folding Vim and VS Code back in explicitly, since they're known quantities from their own courses, alongside every tool this course covers directly.

The Full Roster

- **nano** — minimal, non-modal, zero setup, one file at a time
- **Vim** — minimal, modal, deep once learned (Learning Vim / Vim Intermediate/Advanced)
- **Sublime Text** — lightweight GUI, multiple cursors, near-instant startup
- **VS Code** — lightweight-to-mid GUI, huge free extension ecosystem (VS Code Essentials)
- **IntelliJ IDEA** — full IDE, Java/Kotlin-first, the ancestor platform
- **PyCharm** — full IDE, Python-specific depth
- **WebStorm** — full IDE, JS/TS-specific depth, no free tier

A Decision Framework

- **Task duration/scope** — a single quick edit, or sustained project work over days and weeks?
- **Project complexity** — one file, or a large multi-file, possibly multi-language codebase?
- **Environment** — a local machine with everything already installed, or a remote SSH session with nothing but what's already on that server?
- **Language-specific tooling needs** — does this task genuinely benefit from deep, language-aware analysis, or is it mostly straightforward text editing regardless of the language?
- **Budget** — is a paid IDE actually justified for this specific work, or does a free tool cover everything actually needed?

Applying the Framework: A Few Quick Examples

Editing `nginx.conf` over SSH → **nano** or **Vim**, whichever's already comfortable — task is tiny, environment has nothing else installed. A personal static HTML/CSS/JS site → **Sublime Text** or **VS Code** — no deep language-server tooling genuinely needed. A large Spring Boot enterprise Java backend → **IntelliJ IDEA Ultimate** — project complexity and language-specific depth both justify it. A Python data science project with Jupyter notebooks → **PyCharm** — its own scientific tooling directly matches the task.

TOOL	BEST TASK DURATION	BEST PROJECT COMPLEXITY	BEST ENVIRONMENT FIT
nano / Vim	Quick, single edit	One file	Remote/minimal, zero setup
Sublime Text / VS Code	Short-to-sustained	Small-to-medium projects	Local machine
JetBrains family	Sustained, daily use	Large, complex codebases	Local machine, real setup

EXACTLY WHAT CHAPTER 9 APPLIES NEXT

This framework isn't abstract for its own sake — Chapter 9's own capstone applies exactly these five questions to five full worked scenarios spanning the entire roster, turning this chapter's framework into concrete tool choices.

LEARNING A FULL IDE WELL DOESN'T MEAN USING IT FOR EVERYTHING FOREVER

A real, common trap: once a full IDE is genuinely learned well, firing it up out of sheer habit for a task a much lighter tool would handle just as well — wasting real startup time and resources for no actual benefit. The actual skill this whole course has been building is judgment about fit, not raw IDE proficiency. Being equally comfortable reaching for nano *or* a full IDE, based on the actual task in front of you, is the real goal — not maximizing how often the most powerful tool gets used.

Hands-On Exercises

EXERCISE 1

Using this chapter's own decision framework, walk through which questions matter most for choosing a tool to quickly fix a typo in a README.md file sitting in a large, unfamiliar open-source repository you've just cloned locally.

View solution

EXERCISE 2

A developer who has become highly proficient in IntelliJ IDEA now opens it for absolutely everything, including editing a single-line shell script on their own local machine. Using this chapter's own warning box, explain what's happening here and why it's worth reconsidering.

View solution

EXERCISE 3

Explain why this chapter explicitly includes Vim and VS Code in its own decision framework, even though neither one is taught in depth by this specific course.

View solution

CHAPTER 8 QUICK REFERENCE

- The full roster: **nano, Vim, Sublime Text, VS Code, IntelliJ IDEA, PyCharm, WebStorm** — spanning every course's own coverage, not just this one's
- Five decision questions: **task duration/scope, project complexity, environment, language-specific tooling needs, budget**
- The real skill is **judgment about fit**, not maximizing use of the most powerful tool available
- Chapter 9's own capstone applies this exact framework to five full worked scenarios next

Chapter 8 built a five-question decision framework. This capstone applies it to five genuinely different real tasks, spanning the full roster this course has assembled — closing the course the way Web Servers Fundamentals' own capstone closed that course, with a comparison across worked scenarios rather than a single build.

Scenario 1 — A Production Server Config Fix Over SSH

A single line in a systemd unit file needs fixing, on a remote production server reachable only over SSH, with nothing installed beyond whatever ships by default. Task duration: tiny. Environment: remote, minimal. **nano** or **Vim** — whichever's already comfortable — is the clear fit; neither a full IDE nor even a GUI editor is available here at all.

Scenario 2 — A Personal Static Portfolio Site

A personal HTML/CSS/JavaScript site, no build tooling, no backend. Project complexity: small. Language-specific tooling needs: minimal. **Sublime Text** or **VS Code** both fit comfortably — neither a full IDE's own deep language-server tooling nor a remote-only editor's own constraints apply here.

Scenario 3 — A Cash-Strapped Startup's Growing React/TypeScript Codebase

A small startup's product is growing past a simple prototype into a genuine multi-file React/TypeScript application, but the budget genuinely doesn't stretch to per-seat commercial IDE licenses yet. Budget: a real constraint. **VS Code** fits — genuinely strong JS/TS support via extensions, at no cost, exactly the trade-off Chapter 7 described. If the budget constraint disappeared later, **WebStorm** would be a reasonable upgrade for its own deeper out-of-the-box integration — but it isn't the right call under this specific constraint.

Scenario 4 — A Large Enterprise Java/Kotlin Spring Backend

A large, established engineering team maintains a complex Spring Boot backend in Java and Kotlin. Project complexity: large. Language-specific tooling needs: genuinely deep. Budget: justified at this scale. **IntelliJ IDEA Ultimate** fits directly — Chapter 5's own Java/Kotlin-first depth, plus Ultimate's own Spring-specific tooling, matches exactly what this scale of project needs.

Scenario 5 — A Python Data Science Project With Jupyter Notebooks

A project working heavily with pandas and NumPy, using Jupyter notebooks for exploratory analysis. Language-specific tooling needs: scientific computing, directly. **PyCharm** fits — Chapter 6's own Jupyter integration and interactive DataFrame viewer are built for exactly this kind of work.

SCENARIO	CHOICE	DECIDING CHAPTER(S)
1 — SSH config fix	nano / Vim	Chapter 2 (nano's environment fit)
2 — Personal static site	Sublime Text / VS Code	Chapter 3 (lightweight GUI fit)
3 — Cash-strapped React/TS startup	VS Code	Chapter 7 (budget trade-off)
4 — Enterprise Java/Kotlin backend	IntelliJ IDEA Ultimate	Chapter 5 (flagship depth)
5 — Python data science project	PyCharm	Chapter 6 (scientific tooling)

THE FRAMEWORK DID THE REAL WORK HERE

None of these five choices came from a fixed ranking of "best editor overall" — each one came directly from applying Chapter 8's own five questions to that scenario's own actual constraints. The same framework would produce different answers if any single constraint changed, exactly as Web Servers Fundamentals' own capstone demonstrated with its own decision framework.

THESE FIVE ANSWERS AREN'T A FIXED RANKING

None of these five scenarios crowned one tool "the best editor, period." Each one fit a specific combination of task duration, project complexity, environment, tooling needs, and budget — change any one of those constraints (a startup's budget growing, a personal site adding a real backend) and the right answer could shift entirely. The value of this course was never memorizing which tool wins — it's asking Chapter 8's own five questions freshly, every time.

Hands-On Exercises

EXERCISE 1

Two years later, Scenario 3's startup has grown significantly and now has a healthy engineering budget. Using Chapter 8's own framework, explain whether the original VS Code recommendation should still hold, and why.

View solution

EXERCISE 2

A new scenario: a solo developer is building a small Django web application on their own laptop, with no deployment or team constraints yet. Using this course's own material, recommend a tool and justify it with Chapter 8's own five questions.

View solution

EXERCISE 3

Write a short chapter-attribution summary (2-3 sentences) explaining how this capstone's own scenario-based shape mirrors Web Servers Fundamentals' own capstone rather than Nginx In Depth's single-config build or Android Studio's own narrative session, and why that shape fits this specific course.

View solution

COURSE COMPLETE

Text Editors & IDEs Survey — 9 of 9 chapters complete. This closes out the Developer Tools subject's own current scope.

CHAPTER 9 QUICK REFERENCE

- Five scenarios, five different fits: **nano/Vim** (SSH config), **Sublime Text/VS Code** (static site), **VS Code** (budget-constrained startup), **IntelliJ IDEA Ultimate** (enterprise backend), **PyCharm** (data science)
- Every choice came from **Chapter 8's own five questions** applied to that scenario's real constraints — not a fixed ranking
- Changing any one constraint (budget, project complexity) can flip the right answer — the **framework**, not a memorized list, is the actual takeaway