FROM HELLO WORLD TO TEAM LEAD

A practical guide for developers ready to level up

WRITTEN BY

TIM LORENT

FROM HELLO WORLD TO TEAM LEAD

A PRACTICAL GUIDE FOR DEVELOPERS READY TO LEVEL UP.

BY TIM LORENT

Amsterdam, The Netherlands
www.fromhelloworldtoteamlead.com

To all the ambitious developers out there. This one's for you.

PROLOGUE (NERD FROM THE START)

Two big screens, lines and lines of code, a man deeply focused, paperwork and pencils lying around on the desk. Sounds like an AI-prompt for an image of a developer, but it's actually a description of my father on a daily basis. Code was always around me from a young age. Walking to the kitchen to grab snacks while playing videogames, I'd see code everywhere. Like I was in the Matrix.

Unfortunately, the fascination for the wonderful world of coding did not exactly rub off on me. As a teenager, I rebelled against it. Except for the brief moment I tried to convince my parents to let me drop out of high school to become a developer (thanks again mom & dad for stopping me). Although that was mostly the most valid excuse out I could come up with at the time.

It was for nerds, I told myself. Plus, I honestly believed I was not smart enough to do it. So instead, I had high hopes of becoming a music producer and DJ. I gave it a shot: produced some music, played some gigs. Then I got a 24-hour ringing in my ear and stopped. So it goes.

Fast forward to university, and code came back into my life. This time in the form of a constrictor snake: Python. Boy, did I hate it. I wanted to learn about business and IT, how to connect the two. Why did I have to go through the pain of mandatory exercises in Python, being told not to get too creative? Same story with the web development class: No HTML5, no CSS3. Only older technologies to “really understand everything.” It felt a bit pointless at the time. So when does the part come where everything clicks, I fall in love with code, and we live happily ever after?

Well, that part came years later when I was bored out of my mind during my Master’s in Business Informatics. I absolutely dreaded going to class. But then, out of nowhere, an angel answered my prayers: Turing Society. A free, non-profit coding bootcamp I could follow during evenings and weekends. Eight weeks of frontend web development. I need a new direction in life because the Master’s was not taking me anywhere I wanted to go.

So I gave it a shot. I wrote a few JavaScript functions. It worked. I developed a small website with HTML & CSS. I saw magic on the screen…and fell in love. Before I knew it, I was spending all my free time learning how to code and using all my precious brain power during class to figure out how to refactor my code and write recursive functions.

That was the end of my academic career and the beginning of a new one life filled with creativity, puzzles, complex problem-solving, and honestly, just pure magic. It only occurred to me then: **I was a nerd from the start.**

Six years later, I went from a clueless junior to a team lead. I made mistakes: broke production during lunch, went rogue and started coding outside of the sprint, coded non-stop until I burned out, missed learning opportunities, did `rm -rf` on a production server...no just kidding (for now).

But in the end, I got there and so can you! If you of course follow this guide step-by-step and word-for-word to achieve mega success and make a million dollars! Wait no, that’s not true.

But what I can give you is everything I learned in my seven years of being a developer. Things I applied that helped me grow, improve, and eventually lead developers. I’ve applied this knowledge in my volunteering work, mentoring developers from different backgrounds, and sharing templates and workflows to help them achieve better results.

And I want you to have this knowledge, too.

I want to give back to the developer community and help junior and medior devs (or even anyone who might benefit from this information) achieve great things.

Join me on this journey!

4. TEMPLATES & PROCESSES I USE EVERY DAY

What I’ve often heard is that great developers “just know what to do.” They open a ticket, dive into the code, ship something brilliant, and move on. No checklists. No structure. Just vibes and instincts.

But I am not that developer.

Over the years, I’ve learned that having *systems* doesn’t slow you down—it sets you free. When you build a personal workflow, you make fewer mistakes, collaborate better, and create space to focus on what really matters: solving problems, learning, and delivering value.

In this chapter, I’ll share the actual templates and processes I use every day:

- How I review code
- How I prep for refinement

- The checklist I go through before touching a single line
- My “code commandments” that guide every feature I build
- And the simple frameworks I use to grow, reflect, and keep track of my progress
- The “five why’s” for solving bugs

These aren’t theoretical. They’re real tools I’ve used (and refined) over years of working in product teams, agencies, freelance gigs and tech interviews. My hope is that they’ll help you build your own workflow, one that supports *your* way of thinking and working.

This is the “how” behind my job. Let’s dive in.

HOW I WORK: MINDSET + STRUCTURE

Remember how I described my dad in the preface? Sitting at his desk, surrounded by pencils and paper? He wasn’t just a creative; he was methodical. Every bug, every feature idea, every design thought—it all got written down before he touched the keyboard. His colleagues often asked, “Why write it out when you’ve got a computer?” But that mindset stuck with me.

When I started my first job as a developer, I did the same. At first it was pen and paper. Later it became templates and checklists. And while not everyone works this way—some devs prefer to dive in and figure it out as they go—this system has worked wonders for me. It brings clarity. Focus. Intent.

I've had colleagues tell me they appreciate how structured my approach is. No, you can't predict or plan everything in tech. But you can get *a lot* further by starting with the right questions and a clear map.

That's how I work: with structure that *creates* space for creativity. Systems that support problem-solving instead of getting in the way. I don't just write code—I work with intention, from planning all the way to reflection.

Here's what that looks like:

- I review my "**Code Commandments**" before I open VS Code. Reminders to code for *humans*, not just for the machine.
- I use a **ticket checklist** to make sure I understand the business logic, edge cases, and design choices before writing a single line.
- I follow a **code review template** so my feedback is consistent, thoughtful, and useful.
- I prep for every **refinement** session with a structure that helps me ask better questions and spot unclear stories early on.

Over the years, I've improved each of these tools. I've cut what didn't work. Doubled down on what did. Now I share them with mentees, teams, and clients, and in this chapter, I'll share them with you.

Let's take a look under the hood.

CODE COMMANDMENTS

The 8 principles that guide how I build, fix, and think through code every day. These aren't hard rules. They're reminders. Anchors. Things I return to when I'm stuck, in doubt, or deep in the flow.

1. **Write simple, "dumb" code**: Write your code like the next person who reads it just got hired—and it's their first week. That person might be you in 6 months. Avoid over-engineering. Prefer clarity over magic. If a solution is too clever to explain in a few sentences, it's probably too clever to maintain.
2. **Don't stay stuck for more than 20 minutes**: Frustration ≠ progress. If you're spinning in circles or Googling the same error for the fifth time: pause. Ask someone. Rubber duck it. Walk away and come back. You'll solve problems faster when you stop forcing it. Once I was stuck on a bug and went outside to get an ice cream. The second I got handed the ice cream and started eating, I had a "eureka" moment!
3. **Code should be maintainable and scalable**: **Not just "does it work," but "will it still work next month?"** Your code lives in a system. Can someone extend it without rewriting everything? Does it follow naming conventions? Are the files in the right place?
4. **Know where you're going and how to arrive at your destination.** What does the user need? What's the outcome? What's the flow? Take some time to sketch it out. You'll save hours of rework. I use a checklist before every ticket to make sure I'm not building blindly.
5. **Overcommunicate**: Say more, not less. Leave good commit messages. Comment when something's weird.

Post updates in your team chat. Overcommunication builds trust. No one's ever said, "Wow, you're updating us too clearly."

6. **Know what you know, realise and accept that there is a lot you also don't know**: **Humility = velocity.** You don't need all the answers. You just need to ask better questions. Admitting "I don't know" opens the door to "Let's figure it out." Tech changes fast, but curiosity beats certainty every time.
7. **Use micro-commits**: **One commit, one clean step forward.** Each commit should tell a story: "refactor header," "add unit test". If something breaks, you can easily step back. Here's why micro-commits matter ➜
8. **Ask the five whys**: Get to the root cause. Why did this bug happen? Why didn't the test catch it? Why was it coded that way? Keep asking why—at least five times. Learn the method ➜

Each of these commandments has saved me more than once—from unreadable code, wasted time, embarrassing bugs, or misaligned expectations. You don't need to follow *my* commandments. But create your own. Write them down. Let them evolve. They'll keep you grounded when things get chaotic.

THE REVIEW TEMPLATE

How I approach pull requests with structure, empathy, and clarity.

I used to feel nervous reviewing someone else's code. Who was I to say something should be different? But over time, I realized that a good code review is never about being "right." It's about making the codebase stronger—together. They're a place to learn, to teach, and to build trust in a team.

But only *if* you do them right. So I built a review system. It helps me stay consistent, thorough, and constructive, even on chaotic days.

Let's break it down.

MY REVIEW CHECKLIST

I use this list (or a version of it) in every pull request I review. It's not about ticking every single box, it's about sharpening your focus and asking the *right* questions.

Disclaimer: parts of it are scoped to React, but you can of course change this to reflect your tech stack.

1. Code Structure

- React rendering: Is the rendering logic efficient? Are memoization techniques used appropriately or necessary?

- Hooks: Should certain logic be abstracted into hooks, and are the existing hooks correctly implemented? Should a hook be a regular function instead?
- CSS & HTML: Are correct HTML and CSS practices followed?
    - Can React or JavaScript logic be replaced by native CSS instead?
    - Are any CSS variables skipped?
    - Are styles properly scoped?
    - CSS impact: If a style is removed, what default will the element fall back to?
        - A span with `display: block` will default to `display: inline` if removed.
        - Changing a parent's display from `flex` to `block` could affect layout of child elements.
- State management:
    - Can state be lifted instead of using context unnecessarily?
    - Is the function signature change non-breaking?
        - Adding an optional parameter is fine.
        - Removing or altering required parameters might introduce breaking changes.
- Inversion of control: Does the code follow proper separation of concerns and avoid tight coupling?
- Effects:
    - Is the Effect necessary?
    - Does it describe a single synchronization process?

    - Are all reactive values used in the Effect included in the dependencies array (check with exhaustive-deps)?
    - Is the cleanup logic implemented correctly for scenarios like unmounting or dependency changes?
    - Are objects or functions unnecessarily recreated in the render, causing excessive re-synchronization?

2. Acceptance Criteria Check

Ticket Description: Read the ticket carefully and create a checklist based on its acceptance criteria. Review the implementation against that list.

3. Tests

Coverage: Are all necessary test cases present based on what the component is expected to do?

- Are loading states tested?
- Are error states tested?
- Is the output for different scenarios verified?

4. UI Validation

Responsiveness: If UI was changed, check the branch and manually test it across various screen sizes. Is the UI fully responsive?

5. Accessibility (a11y)

- Accessibility Guidelines: Are the applicable accessibility rules followed?
- A11y checklist: Has the accessibility checklist for the project been consulted or completed?

Additional Review Considerations

- Impact of changes: Always consider how far-reaching a change might be.
    - When modifying a function or component's signature, think carefully about breaking changes.
    - When changing styles, consider how fallbacks might affect layout or accessibility.
- Data handling: Consider the kinds of data this function or component might receive.
    - How does the implementation behave with edge cases?
    - Are there any assumptions that could break with unexpected input?

REVIEW PHRASES THAT HELP

Here are a few lines I often use to keep feedback clear, respectful, and collaborative:

- “Would it make sense to…?”
- “What do you think about this alternative?”
- “I wonder if we could simplify this by…”

- “Totally optional, but I’ve seen this pattern used effectively here...”
- “Just a preference—feel free to push back!”
- “This is looking good. One thing I’d consider...”

And just as important: I **never** write vague or dismissive comments like:

- “This doesn’t look right.”
- “Fix this.”
- “Why is this here?”

Why not? Because it puts someone on the defensive and doesn’t invite a conversation.

HOW I GIVE FEEDBACK

Here are my three golden rules:

1. **Be specific.** Don’t just say “this needs work”! Say *why*, and offer an example.
2. **Be empathetic.** Someone spent time on this. Even if something could be improved, there’s usually something to appreciate too.
3. **Be collaborative.** A review isn’t a monologue. I often ask: “What are your thoughts?” This opens the door for discussion instead of judgment.

I also always assume good intent. If something looks off, I first check the context, and I ask—not accuse.

WHY REVIEWS MATTER

A good PR review isn't just about finding bugs. It's a chance to:

- Strengthen team knowledge
- Share patterns and best practices
- Prevent future tech debt
- Help someone feel more confident in their work

When I first started out, I noticed something interesting: senior developers would often spot things in my code that I completely missed. Not because I wasn't paying attention, but because they had a broader perspective. They would ask questions about architecture, reusability, long-term impact—things I hadn't learned to consider yet.

I thought: I want to review like that.

So I did two things:

1. I asked them directly how they approach a code review. What they look for, in what order, what questions they ask themselves.
2. I studied their comments. I went back through their reviews, one by one, to see how they thought, how they phrased things, and how they gave feedback.

Then I started applying those techniques to my own reviews. Little by little, my comments became more strategic, more helpful—not just for the author, but for the whole team.

Code reviews are one of those places where your technical skill and your people skills meet. Be clear, be kind, be curious.

And next time you review a PR, ask yourself: *Would I feel good receiving this comment?*

THE TICKET CHECKLIST: START SMART, FINISH STRONG

You know that feeling when you're halfway through a ticket and suddenly think: *wait... what exactly am I building again?* Been there. And trust me, it's not a great place to be. That's why I started creating a personal checklist before touching any code.

For me, the real coding doesn't begin when I open VS Code. It starts *before* that, when I understand the assignment fully, identify the risks, align with backend and design, and make sure I'm not just "building stuff," but solving the right problem.

This section isn't just about planning. It's about building intentionally. Checklists don't make you rigid. They make you focused.

They help you:

- Understand the scope
- Catch red flags early
- Ask better questions
- Avoid rework or misunderstandings

It's also a huge time-saver. Not just for you, but for your team. When you know what to ask *before* you start, your work becomes smoother, faster, and more aligned.

MY TICKET CHECKLIST (SIMPLIFIED VIEW)

Here's the core structure I follow before and during any ticket:

1. Prep

- Read the ticket thoroughly
- Review the project's guidelines
- Skim my own "Code Commandments"
- Ask: Is the API ready? Do I understand the business logic? Is the scope clear?

2. Requirements

I write out:

- The **goal** of the ticket
- The **context**: why it matters
- Links to designs, docs, or related tickets
- A numbered list of **acceptance criteria**

3. Questions to ask

- **Design**: What happens on hover? Are there empty states? Do we need animations?

- **Accessibility**: Is this element focusable? Is there keyboard navigation?
- **Technical**: Is the data model clear? Do I know what I'm sending/receiving?

4. Coding

- Micro-commits only
- Document decisions inside PRs
- Sync with backend if something's unclear

5. Testing

- UI tested on various screen sizes
- Manual checks with dev tools (e.g. throttling, a11y scanners)
- Unit & integration tests written
- Acceptance criteria validated

6. Delivery

- Review the PR carefully before opening
- Link back to the ticket and include a clear summary
- Delete branch after merge

What it helps with (real examples):

- **Once, a ticket looked simple. A** new UI component. But after writing out my checklist, I noticed the design had no loading state. I flagged it before writing a line of code. The designer updated the file and thanked me. That saved us a full feedback round.
- **Another time, I almost started coding with the wrong API endpoint**. My checklist forced me to double-check

with backend. And guess what, the endpoint wasn't stable yet.

These aren't big hero moments. But they're the quiet decisions that make or break a clean delivery.

ASK BEFORE YOU CODE: THE MINDSET

It can be tempting to jump straight into solving. I get it, building is fun! But I've learned that slowing down at the start *actually* helps me speed up later.

That's what this checklist is about: catching ambiguity, reducing back-and-forth, and being a good teammate.

Here's the rule I follow: If I can't clearly explain what I'm building and why, I'm not ready to code yet.

MY STORY REFINEMENT FRAMEWORK: SHOW UP PREPARED

Refinement sessions aren't just about "estimating the work."

They're about making sure *we all understand what's being built*—and why. Over the years, I've noticed that the developers who get the most out of refinements… are the ones who don't walk in blind.

That's why I always block 30 to 60 minutes *before* a refinement session to go through the upcoming tickets on my own. No distractions. Just me, the stories, and some water (no, I don't drink coffee).

Why prep matters? When you've already reviewed the stories in advance:

- You ask better questions
- You spot gaps before they become blockers
- You help your team move faster

It also shows your team you care—that you're not just there to passively nod along, but to co-own the outcome.

MY REFINEMENT CHECKLIST

Before the meeting, I go through each ticket and jot down notes like:

- **Ticket.** Jira ticket URL
- **Goal.** What's the user need? Is that clearly stated?
- **Scope & Assumptions.** What's in scope and what's *not*? Any risky parts? Do I need clarification from product or design?
- **Design Questions.** Are all states clear (hover, error, loading)? What happens on mobile?
- **Dependencies.** Does this ticket rely on backend work? Is the API even ready?
- **Estimation.** What's a realistic range? I usually write: 4h, 8h max (with caveats)

What I do when a story is vague or messy? If a ticket is unclear, too technical without context, or missing design:

- I *don't* just flag it during the session

- I write down my questions and ping the right person ahead of time
- I suggest reframing the ticket (e.g. splitting, adding acceptance criteria)

You don’t need to solve everything solo, but you *do* need to drive clarity.

BONUS: WORKING WITH DESIGN & PRODUCT

Good refinement doesn’t just happen in dev-land. I regularly loop in:

- Designers: to walk through specific screens or edge states
- Product Owners: to confirm business rules or explain context
- Testers: to check that we’re building something testable and traceable

Refinement is a team sport. Show up ready to play.

FRAMEWORK FOR GROWTH

At some point in your career, your growth won’t come from tutorials or coding challenges anymore. It’ll come from reflection. From looking back at what you’ve built, how you’ve contributed, where you’ve stepped up, and what you’ve learned along the way.

That’s why I started building my own growth framework. Not to impress anyone—but to stay honest with myself.

And it works.

Every year, I sit down and I reflect: *Where did I show impact? What do I want to improve next?*

I split this into three layers: **Role Expectations**, **Impact**, and **Achievements**. Simple, but powerful.

1. ROLE EXPECTATIONS

Start with your current level (junior, medior, senior, lead). Then write out what’s expected of you—not just the tasks, but the mindset.

Here’s a quick medior-level example:

- Can work independently on most tasks
- Acts as a mentor for juniors
- Spots opportunities to improve the codebase or process
- Suggests direction for the frontend
- Conducts small-scale research and gives technical advice
- Can structure and break down complex tickets
- Writes clear, maintainable code
- Reviews and improves the work of others

Your version may look different. The goal is clarity: *what does “doing well” look like for me right now?*

2. IMPACT

Ask yourself:

- What impact have I made on my team?
- Where did I improve the process, product, or performance?
- What feedback did I get (good or bad)?
- Where did I show initiative?

This can be high-level, like “led frontend refinements” or “created onboarding docs for new team members.” Or it can be subtle, like “started asking more questions in sprint reviews and helped catch two issues before go-live.”

Write down both your **wins** and your **growth areas**. Don’t filter. Growth starts with awareness.

3. MONTHLY ACHIEVEMENTS

Every month, I write down key things I achieved, learned, or contributed to. Even the small stuff counts. Especially the small stuff.

Here’s a real example from my logs:

June 2024

- Wishlist sync feature built with a reusable architecture
- Introduced a11y checks to the story-writing process
- Contributed to Q2 planning by reshuffling priorities with the BA team

- Presented accessibility improvements during frontend temple
- First customer-facing demo given to global stakeholders

Pro tip: Don’t wait until the end of the month. Keep a running list.

Write things down *as they happen,* even just a line in a notes app or Notion doc. You’ll be surprised what stacks up after a few weeks! One moment you’re just logging a small fix or presentation… and suddenly, you realize you’ve shipped five features, mentored a teammate, and organized a workshop. It’s easy to forget how far you’ve come unless you keep track. And when it’s time for a review, promotion, or portfolio update—you’ll have everything ready to go.

MISSED OPPORTUNITIES

Just as important: note what you could’ve done better. What feedback did you ignore? What initiative didn’t you take? What fear held you back?

My 2024 log includes things like:

- “I should’ve asked to be considered for a lead role earlier.”
- “Need to speak up more during performance-related tickets.”
- “Let a11y slip in a few PRs—need to double down again.”

This isn't about being hard on yourself. It's about growing with intention.

TRY THIS: BUILD YOUR OWN GROWTH TRACKER

Start today with this template:

Section	Example
Level	Medior Developer
Expectations	Independent delivery, mentor juniors, initiate improvements
Impact Areas	Frontend processes, a11y, onboarding
Monthly Achievements	Write down 3–5 per month
Reflection Notes	Wins + What to do differently next time

You can do this in Notion, a journal—whatever works. The key is: make it yours. Make it a habit.

Why This Works

- You don't forget what you did
- You make your value visible (to yourself and others)

- You learn faster because you stop repeating old mistakes
- You're never stuck in a review going "uhhh... I don't know what I did last quarter"

Your career is made of hundreds of micro-moments.
If you don't write them down, you'll forget how far you've come. This is how you stay grounded, confident, and intentional. This is how you grow.

TOOLS I ACTUALLY USE

This chapter has mostly focused on mindset and systems, but let's get a little more practical. Over the years, I've tried dozens of tools, productivity hacks, to-do apps, and workflows. Most of them didn't stick. But a few became daily drivers.

These aren't just shiny tools I downloaded once. These are tools I've actually used for *years*, refined over time, and integrated into how I think and work. Some are technical. Some are organizational. All of them serve one purpose: to reduce friction and help me stay intentional.

Thinking Tools

Bear
This is my second brain. I use it for literally everything:
- My code commandments
- Project notes
- Template library

- Achievements list
- Reflection notes after projects
- Meeting prep

It’s fast, clean, markdown-based, and works offline. I’ve tried Notion, but Bear feels lighter and more flexible for personal workflows.

Things
My weekly planning and accountability hub. Each week, I list:
- Must-do’s
- Nice-to-haves
- “Don’t forgets” (small tasks that sneak up on you)

Every Friday I review what got done—and what didn’t. Things helps me build a rhythm without feeling like a micromanager to myself. At the start of each week, I open Things and ask: “What’s actually important this week?”

Pomodoro Method (25/5 and 50/10)

I don’t always need it, but when I do it works wonders.

- **25/5** when I need short bursts of focus, e.g. ticket analysis or creative thinking
- **50/10** when I’m deep in the code and want to stay there

It’s not about the timer—it’s about protecting your focus. Especially in open-plan offices or busy teams.

Mental Frameworks (Structuring How I Think)

Tools are more than apps. These frameworks changed how I work:

- **GRIP (Rick Pastoor):** Practical system for planning your week, managing meetings, and keeping your inbox (mostly) sane
- **Deep Work (Cal Newport):** A must-read for anyone who writes code or solves complex problems for a living. Less noise = more quality
- **The Five Whys:** My go-to method for debugging and understanding root causes
- **Weekly Reviews:** Every Friday, I look back: what worked, what didn't, and what I want to try next

THE WEEKLY REVIEW: RESET, REFLECT, REFOCUS

I'll be honest: I haven't done this every week. There were periods where I completely dropped the habit. But every time I return to it, I realize just how powerful it is.

SMALL STEPS EACH DAY—1% BETTER

The weekly review is my personal reset button. It gives me space to reflect, reconnect with my goals, and plan the week ahead with more intention.

I don't just use it to check off tasks. I use it to ask:

- What gave me energy this week?
- What drained me?
- What did I actually learn?
- What do I want to focus on next?

It's also where I evaluate open projects, clean up my digital desk (yes, that downloads folder), and curate what I want to read, watch or explore next in frontend. I block time for learning. I ask what would make the week *great*—not just productive.

Here's what a typical weekly review looks like for me:

Weekly Review Checklist

1. Reflect on last week: what worked, what didn't? What gave you energy? What drained you?
2. Clean up desktop & downloads
3. Review calendar & meeting notes. Any follow-up actions to take?
4. Process inboxes & task lists
5. Check open projects. Are tasks still relevant? Is there a next action defined?
6. Revisit goals. Any progress? What's the next step?
7. Plan your week. Choose 2–3 key priorities. Block time for deep work, learning, and rest

Weekly Journal Prompts

- What would make this a *great* week?
- How will I work on my growth this week?
- What article or video will help me stay current?
- What will I do differently this week?
- Which goals deserve extra focus right now?

Do I follow this checklist perfectly every time? Absolutely not. But even a 15-minute version of this ritual gives me clarity and confidence going into a new week.

Pro tip: Don't wait until Sunday night. Find a moment that works for *you*. For me, Friday afternoon is perfect: I close the loop on the week, clean my mental workspace, and actually *look forward* to Monday.

Give it a try. It's one of the highest ROI habits I've built — and it costs you nothing but 30 quiet minutes and a little honesty with yourself.

The Weekly Review: A Productivity Ritual to Get More Done

INTEGRATING TEMPLATES INTO YOUR DAY

Let me be real with you: I don't use all these templates every single day. I'm not a robot. Sometimes I just want to fix a bug, write some code, and skip the checklist. And that's okay.

Templates are here to help you, not to control you. They're tools, not rules. You're allowed to break them. The point is not to become a productivity machine, but to work with more clarity, confidence, and intention when it matters.

That being said, here's how I *do* use them in practice:

- **Start of the day**: I glance at my *Code Commandments* before I open my editor. It's a quick reminder of what I value: readable code, micro commits, asking the five whys.
- **Before a ticket**: I run through my *Ticket Checklist* to make sure I understand the scope. Not always in full detail, but even scanning the headings helps me spot missing context.
- **Before a refinement**: I block 30–60 minutes to go over the stories. I prep questions, assumptions, edge cases. It makes the meeting 10x more valuable.

- **During a PR**: I open my *Review Template*. Especially if I'm tired or busy, it helps me zoom in and out: structure first, then details, then UX, then accessibility.
- **End of the week**: I don't always do a full weekly review, but when I do, it feels like clearing mental RAM. I reflect, clean up, and plan with a fresh mind.

What I've learned is this: **having a process doesn't mean you follow it blindly**. It means you have a foundation. Something to return to when things get chaotic. Something to build on when you're ready to grow.

If you want to create your own templates:
- Start with what already works for you
- Write it down, try it out, and update it regularly
- Don't overthink the format, just make it usable
- Ask teammates what *they* use, and steal the best ideas

And most of all: **don't let it become a cage**. Your process should evolve with you. Skip the checklist when you're in the zone. Come back to it when you feel lost. That's how you make systems that *serve* you, not the other way around.

KEY TAKEAWAYS

- Having a system doesn't kill creativity, it supports it. Templates help you focus on what matters, reduce mental load, and stay intentional.
- A checklist or template is a guide, not a rulebook. Use them when they help. Ignore them when they don't. Make your process work *for* you.
- From achievements lists to weekly reviews, looking back helps you move forward. Keep track of what works, what doesn't, and where you want to go.
- Whether you're writing a commit, reviewing a PR, or refining a story: clarity, empathy and curiosity go a long way.
- Great developers don't just write great code. They have reliable workflows, clear priorities, and systems that help them deliver.

NEXT STEPS/REFLECTION

Try one of these actions this week:

- Write your own **Code Commandments**. What values do *you* want to bring to your work?
- Use the **Ticket Checklist** before starting your next story (even if it's just a quick scan).
- Create a **Review Template** based on how *you* give feedback. What do you wish someone had told *you*?
- Start a weekly **Achievement Log.** What did you ship, solve, or learn?
- Run a short **Weekly Review** and plan your next sprint of personal growth.

In the next chapter, we'll zoom out. You'll learn how to take ownership, step up in your team, and grow from junior to senior (and beyond). Not just by writing code, but by showing up with the right mindset, habits and momentum.

Because growth doesn't happen by accident.

It happens when you build it into your way of working.

Let's go!

WANT MY FULL WORKFLOW?

This chapter is a sneak peek into how I bring structure and clarity to daily technical work. In my eBook *From Hello World to Team Lead*, I go deeper into:

- My full set of templates for reviews, refinement, and reflection
- Checklists for performance reviews and growth tracking
- Small developer habits that drive big impact

Ready to level up your workflow? You've got two options:

1. **Grab the eBook** – with all templates included as a bonus
2. **Book a free intro call** – and explore how I can help you build a system that fits your style.

👉 From Hello World to Team Lead

👉 Book a free call

Let's build the career you want.

ABOUT THE AUTHOR

Tim Lorent. I'm a frontend developer, mentor, and coach. Over the past 6+ years I've gone from bootcamp student to lead developer. Now I help other developers grow with clarity, intention, and practical tools, through coaching, content, and community. You can find me here:

→ https://www.fromhelloworldtoteamlead.com/
→ Tim Lorent - LinkedIn