JOSEPH MAXWELL

The Art of Ecommerce Debugging

An entertaining guide to quickly and effectively solve problems for ecommerce developers.

FOREWORD BY ERIK HANSEN

The Art of Ecommerce Debugging
Copyright © 2021 by SwiftOtter, Inc.
Printed in the United States of America.

All rights reserved. No part of this book may be reproduced in any form or by any electronic or mechanical means including information storage and retrieval systems without permission in writing from the publisher, except by a reviewer, who may quote brief passages in a review.

Published by:
SwiftOtter, Inc.
Phone: 1-888-57OTTER
Web site: swiftotter.com

Visit swiftotter.com/artofdebugging where you will get access to an exclusive community of debugging experts.

1
ISBN 978-1-7367586-0-1

JOSEPH MAXWELL

The Art of Ecommerce Debugging

An entertaining guide to quickly and effectively solve problems for ecommerce developers.

SwiftOtter

KIND WORDS

This book is fun and fast to read, and I picked up some tricks I didn't know before (after 24 years). Besides the tricks and stories, it's also very motivating to keep growing as a professional developer! I would consider it a good gift to any PHP developer, especially in the ecommerce world.

Vinai Kopp, Magento expert

Debugging PHP applications has come a long way since `var_dump()` and `die()`. If you are still stuck in those days, let Joseph show you not only the new tools, but the strategies you need to debug today's complex web applications.

Cal Evans, World-famous PHP developer

This book is a journey through the thought processes of debugging and solving development problems. It's the ultimate dissection of the mind and thought process of a master skilled ecommerce development architect. So many of these things developers just have to learn by experience are explained clearly and concisely. It's going to be part of our required reading for new developers at Netalico, and I believe it will be a future staple in every ecommerce developer's education tool kit.

Mark Lewis, CEO of Netalico

In his book, Joseph Maxwell answers the voice of Magento developers, shares valuable knowledge and his passion and experience. It's not another tutorial book but rather a journey that changes your mindset on being an expert.

Lukasz Bajsarowicz

I grew up on coding during the early 2000s, in an age of few references and all-new problems to solve. Counting on a book like *The Art of Ecommerce Debugging* would have been the missing stone for a boosted learning and a "not-alone" troubleshooting.

It will surely be the next magic card to use with my team and to suggest to my colleagues.

Salvo Capritta, CTO of Synthetic Lab

Thank You!

I want to thank my friends who took the time to review this book and provide insightful comments as to how I could make it even more helpful for you. These are smart people, so if you come across them online or in person, make sure to learn from them.

Erik Hansen
Alessandro Ronchi
Lukasz Bajsarowicz
Silvia Pistore
Russell Albin
Michael Manreza
Sander Mangel
Kliment Ognianov

I owe my greatest debt of gratitude to my wife, Elissa. She has supported and encouraged me through this entire project. She is my greatest cheerleader even when I felt like giving up.

Simon Frost
Helal Uddin Patwary
James Cowie
Manju Chauhan
Ruggero Landolina
Jackie Angus

CONTENTS

FOREWORD

Early in my web development career, I spent countless hours trying to solve various problems, often to find that the solution had been staring me in the face, had I only known where to look. I once took down a large ecommerce site and spent hours frantically trying to solve the problem. It turns out I inadvertently deleted a single row in a database table that caused the site outage. In the years that followed, the school of hard knocks helped teach me a lot of valuable debugging skills. If I could send this book back in time, I could have avoided many of those pitfalls.

Joseph Maxwell is a highly recognized developer in the Magento community and is a Magento 2 core contributor. He has achieved the highest honor given by Magento in 2019: the Magento Master designation. His study guides have helped thousands of developers get their Magento certifications. In this book, he provides a framework for effective debugging, along with many tools and strategies for how to debug more productively.

I've worked with developers from various backgrounds and found that many developers who are great at *writing* code don't necessarily know how to effectively *debug* code. Whether you're a new developer or have been writing code for years, this book should set you on a track to master the art of debugging.

Erik Hansen

CEO & Founder, Kraken Commerce

OVERVIEW

Mmm... Spaghetti from that new restaurant!
Umm, actually... this doesn't taste good at all.
What do we have here? Uncooked tomatoes, some half-frozen noodles...
The chef definitely skipped QA — and may need some 'programming help'!

Introduction

Perhaps you have experienced a situation similar to this. After a long day of work, you arrive home and sit down to eat dinner. On the menu, spaghetti, one of your favorite meals! You inhale your first mouthful when you see something strange: carefully hidden under the thick layer of red sauce is a pile of green, leafy spinach. *What?* You get over the surprise only to be on to the next one: pale, yellow strings resembling noodles, only they're not. You now remember your spouse is on a health kick, and these strings are actually spaghetti squash – a fruit that bears a striking resemblance to spaghetti noodles, but only in looks, certainly not in taste.

In software development terms, "spaghetti code, but with a nasty surprise" is an excellent description of the code I produced for the first 10 years of my journey. This code was a blast to write but a nightmare to debug.

Through these pages, I share my findings and personal experiences on my journey as an ecommerce developer. I even divulge my worst missteps – all in an effort to kickstart your journey as a developer and help you prevent the same problems I encountered.

I estimate my seniority through the first 10 years of my software development journey, at best, as an upper-level junior developer. Why? Surely 10 years would catapult me to senior ranks? No. I didn't have principled methodologies for solving problems. I got tunnel vision. I had limited tools.

However, growing up as the lone developer laid the foundation for me to grow the character necessary to effectively solve problems. The framework and tools came later. This character, framework, and tools are all discussed in this book.

My story and gratitude

I started writing code in the late 1990s, at 10 years old. My introduction to writing code was to Object Pascal and the Borland Delphi IDE. On my 12th birthday, my parents gifted me with something I imagine not many other 12-year-olds would like (but I certainly loved it!): TurboPower's Orpheus component pack. This was a set of tools to help build programs faster.

Around this time, Dave Klein came into my life. He was one of the kindest and most generous people I'd ever met and also a master of Delphi. While my interactions with him were somewhat rare, he was my hero. He gave me ideas on how to solve a number of my difficult problems.

Next, I moved into Visual Basic and C#. Can you believe it, I got a Dell Axim for Christmas one year – so I built an app I dubbed TimeTracker Pro. I also wrote an app for a local construction company – to assign workers to job sites. Shortly thereafter, I moved into the web world and created a few applications with C#. I chose not to use the built-in MVC pattern as that sounded awfully complex. Instead, I wrote every page as its own .aspx file; each file was chock-full of spaghetti code.

PHP was the next language to tackle, and Dave Whitinger came into my life. I kid you not, this guy probably had at least a 160+ IQ score. He was a Linux genius and ran a well-known Linux news website – but he also wrote PHP. It blew my mind just how quickly he could whip up a program, and how kind he was to also give me advice.

The brainchild of my initial efforts in PHP was a "shopping cart." What an accomplishment! It transacted over $100,000 of revenue. You might be able to be slightly impressed so long as you didn't look at the code or troubleshoot it. More bugs lived in this codebase than in a bug farm specifically designed to grow bugs.

2011 brought along a new platform for me: Magento. Another hero arrived: Lee Saferite. Lee, a student of best practices (not to mention a brilliant and *very patient* fellow), was a developer who knew the Magento platform inside and out. While working with him, he gently called out some bad habits I had developed and gave me advice.

Thank you to my heroes. *The Art of Ecommerce Debugging* is my attempt to pay it forward.

Who is this book for?

This is for Joseph, 20 years ago. I wish I had something in book format that laid out tools I could use to grow and solve problems quickly.

As such, *The Art of Ecommerce Debugging* is **for you**. I hope to accelerate your journey to climbing the echelons at your company through your ability to troubleshoot. I aim to lead you into writing code and fixing errors so you won't have to describe your code as "spaghetti code, with a nasty surprise."

While I write this from a background in Magento, this book is for all who develop ecommerce applications (Magento, WooCommerce, BigCommerce, Shopify, etc.).

We are a special breed of developers. Our applications are huge with steep learning curves. We face very unique challenges. We are under significant pressure to maintain uptime.

Take a moment to register on swiftotter.com/artofdebugging. You will get access to an exclusive community of debugging experts. Share your problems and also your answers with others. You will get it all here.

The 30,000-foot view

There are three parts to this book.

Part 1 goes through the theory of what it takes to be a great ecommerce developer. I hope to prove you are *not limited by anything* – except to the extent that you believe you are limited. Limitations are figments of your imagination.

But why is this included in a book titled *The Art of Ecommerce Debugging*? Very simply, if you are not committed to being a great developer, you will in no way master the ability to troubleshoot problems. I firmly believe our excellence is rooted in our character, and we must start there.

Part 2 goes through my framework for debugging problems. This is a super practical conversation. You will learn a specific framework in which you approach problems and solve them. The primary goal is to give you timeboxes within which you work – this relieves stress and prevents tunnel vision.

Part 3 discusses strategies and tools that are at your disposal to equip you to solve problems quickly. While you might be tempted to skip ahead to Part 3, please don't. You first need to build the foundation – with Parts 1 and 2 – on which you can solve problems. Then, Part 3 will be your toolset to finish it out.

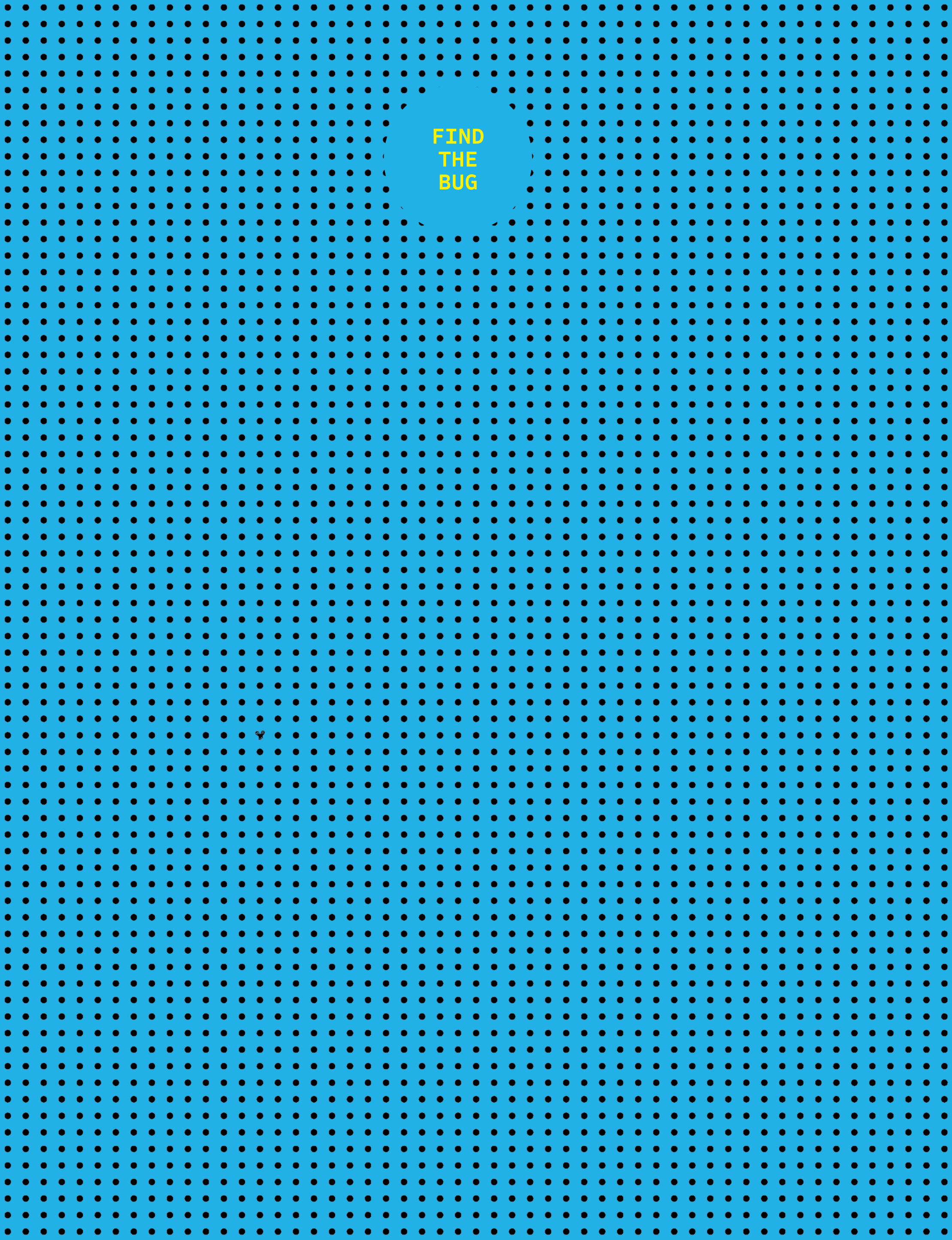
"But, Joseph, I don't have the memory capacity to store all this information, let alone every solution I've come up with! How am I supposed to do this?"

I want to make three important points:

1. You can't remember everything. I find great developers learn general frameworks for problem solving (we will talk about the TAD framework in this book) as well as for solutions. While they don't remember all the specific details, they know where to find those missing details.

2. You must continually practice. The ideas in this book will take time to implement. No one earns their black belt overnight. Years of hard work and training climax in such an accomplishment. Your journey is the same. Chapter 3 discusses continual improvement, which is key to your success as a great developer.

3. While many developers hate documentation, it really is your friend. You won't forget anything you write down ... unless you lose that piece of paper or file.


FIND
THE
BUG

Part 1

A Great Developer Is Key To Great Solutions

FIVE RESOLUTIONS ON WHAT IT TAKES TO BE A GREAT DEVELOPER

Woah, who's that guy strutting around with a scepter and cape?
That's a Senior Developer.
Does he know he's shirtless?
He doesn't care!

Is being a senior developer the zen of development?

Senior developers are known to make significant money. They get to work on fun and challenging projects. They might even manage other people. What could be better than this?

Definition of a senior developer:

- Has many years on the job.
- Has complete independence. They don't need to be told how to solve a problem.
- Has tools to solve problems quickly.
- Has a large repertoire of solutions for problems. We call this experience.

(Sadly, many who have the title "senior developer" cannot pass even basic proficiency tests – they have the title only because they have been writing code for five to seven years.)

Bummer. "So I need to spend seven years working my tail off to become a senior developer?" Unfortunately, yes – there is no way to transfer and implant experience in your head. This is why senior developers are valuable and paid big bucks.

But there is good news. I would argue a significant subset of a senior developer's value includes the *skills* they bring. Skills **can be** transferred, learned, and put into practice – immediately. Welcome to *The Art of Ecommerce Debugging*.

While we all likely aspire to be a *senior* developer, I propose there is a higher and more noble calling – that of a **great** developer. This is a pun borrowed from the book *Good to Great* by Jim Collins.

Great developers are on a journey to which there is no end but many significant rewards to gain along the way. In fact, Ernest Hemingway said that "we are all apprentices in a craft where no one ever becomes a master." A great developer can also be a senior developer. A senior developer is not necessarily a great developer.

Senior developers command a significant paycheck. The value of a great developer far exceeds the amount of his or her paycheck.

In the upcoming chapters, and in the context of debugging, I will define a great developer as someone who:

- Never gives up.
- Methodically solves problems – with flexibility.
- Constantly learns new ideas.
- Reports back early and often.
- Builds debuggable applications.

As developers, we often like to focus on our technical prowess (hard skills): these are comparatively easy to learn. I would argue these soft skills of character, mindset, and the ability to troubleshoot are far more valuable to employers. While technical knowledge of a platform can be easily taught, your willingness to develop these soft skills is something only you can do.

Did you notice none of these points are related to years on the job? That's great news! This book is to present you with the tools and plan for how to accelerate your arrival as a senior developer – and that is through becoming the greatest developer you can be.

You are only a few steps away from being a developer who is truly excellent in what you do.

To be clear, there are other important points to being a great developer. My goal is to cover the ones that are critical to being excellent in debugging.

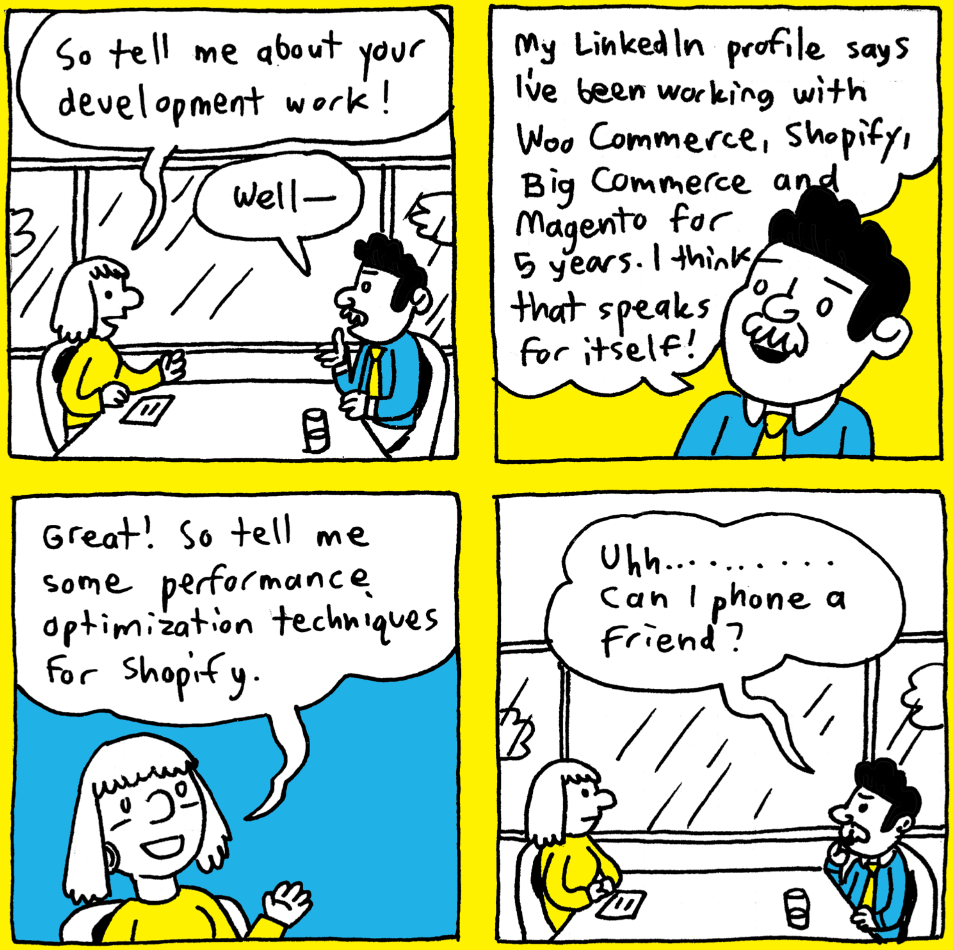
The first five chapters will investigate each of these characteristics.

Shouldn't we have another description of a great developer titled "Trains and educates others in this role"? Giving back is very important. Once you master a skill, pass it on. Invest in others. Share your knowledge with your teammates. Write blog posts and contribute to online forums. Speak at conferences. The more you educate others, the more you also grow and refine your knowledge.

That's why I wrote *The Art of Ecommerce Debugging*—to pass on to you what I have learned and have been taught by others.

Chapter 1

NEVER GIVES UP.



So tell me about your development work!
Well—
My LinkedIn profile says I've been working with WooCommerce, Shopify, BigCommerce and Magento for 5 years. I think that speaks for itself!
Great! So tell me some performance optimization techniques for Shopify.
Uhh.......... Can I phone a friend?

A great developer never abandons a challenge and will fight through to the completion.

There I sat at my desk, a newly-minted teenager with a prominent cowlick in the front, and one in the back (earning a nickname of “rooster tail”). I was hard at work on my “programming” project: a time-tracking software. It was dubbed *TimeTracker Pro* (“Pro” sounded more official for some reason). This name might not have been so off-track after all, as I now see software with this very name.

My new project seemed promising, and it worked quite well, *until* my first beta tester uncovered a nasty bug.

First, we must look at my architecture of this “app.” The user selected the billing client and clicked Start. I used a timer component, which triggered an event every second. I added this second to an ongoing tally of seconds in which this task was open. Once the user finished, she clicked Stop and the event stopped triggering. Sounds good? Well, I thought so, but here was the problem.

When the mobile device turned off, the timer stopped incrementing. This resulted in significantly under-tracked time.

I worked my tail off to find a solution – for days. Maybe I could have the timer run in the background? Maybe I could try to watch for when the timer was turned off by the system?

All of this was, according to an English idiom, “barking up the wrong tree.” I focused on a specific set of solutions, never realizing that the solution was something completely different. Yet, I determined I would stop at nothing short of the fix as I had no team or senior developer to whom I could pass off this project.

I looked for help outside of my skin: an Internet forum (where was StackExchange?!). Someone kindly replied with:

> *What about viewing time tracked as the difference between the current time and the start time?*

Instead of treating time as additive by tallying seconds, I calculate the difference between “now” and when the timer started. *This turned out to be the solution to my problem.* This worked whether the device was on or off.

I did not achieve the solution through calculating time via a different algorithm. The solution came through not giving up and asking the right people. This solution ended up being a completely new way of approaching the problem. Thank you to the “random Internet user” who helped me.

Joel Spolsky, one of the greatest minds in software development, wrote an article called the “The Guerrilla Guide to Interviewing.” One of his premises is that he looks for developers who are Smart and Get Things Done. Are you someone who stops at nothing to get a job done – and right?

Herein lies the most significant trait of a great developer: they have learned no task is out of their reach. If something is difficult, the great developer communicates they need more time. And they make this known early and often. They are also willing to ask for help. They have learned the tools to effectively reach completion (see Parts 2 and 3) while also *accepting the responsibility* to see a task through to its finish.

How this is lived out

1. Self-assessment

I'm sure you regularly encounter difficult problems because that's the name of the game for us as developers. But remember there is a world of difference between those who ask for advice (after communicating what they've tried) and those who take any opportunity to pass off the problem assignment. The former is very good. The latter is ... well, not a mark of a great developer.

The best way to analyze yourself is to look at your past year of performance and ask:

- How many times did I ask someone else to solve a problem assigned to me?
- On the other hand, how many times did I spin my wheels trying to solve a problem when I could easily have shared my findings with someone else and asked for their advice?
- How many issues were re-assigned to someone more senior because I did not have the skill level to complete them?
- How many times did I ask a member of my team a question that could have been easily answered, for example, through a Google search or opening a file?
-

2. Define the task

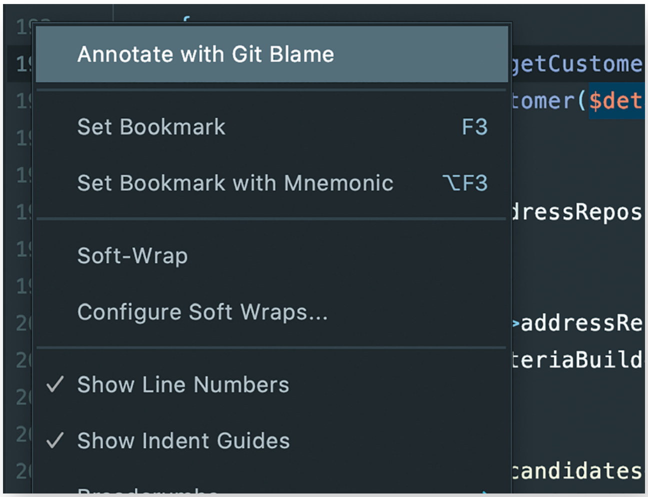
Let's say your project manager assigns you a ticket where orders are not being transferred into the merchant's ERP system. You are not familiar with this integration.

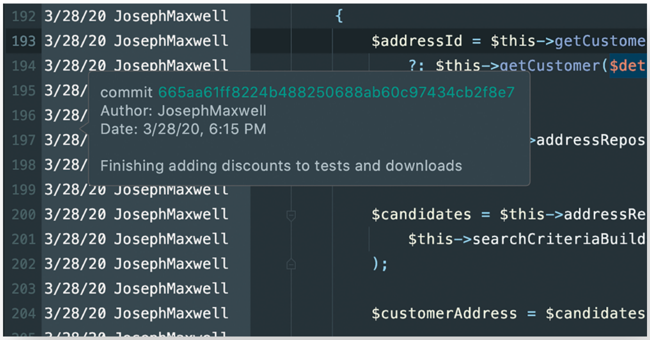
First, the ticket likely has fewer details than necessary to fix the problem. Add the details. Make the task understandable. Here are some questions you should ask yourself as you begin:

- How does this fit into what I know about the merchant's website?

- What assumptions am I making?
- What problems will this introduce in the future?
- Do we have a solid estimate for this change?
- What is the impact on other parts of the system?
- What business objective is this supposed to accomplish?

When you come up with details that are yet to be answered, investigate them yourself. Few merchants, except for the most cost-conscious, wish to repeatedly answer the same questions. Many questions can be easily resolved by looking through prior ticket history, email exchanges, or the codebase, especially when commits are tied to ticket number, like this:





```
3/28/20 JosephMaxwell        {
3/28/20 JosephMaxwell            $addressId = $this->getCustome
3/28/20 JosephMaxwell                ?: $this->getCustomer($det
3/28/
3/28/
3/28/                                              >addressRepos
3/28/
3/28/20 JosephMaxwell
3/28/20 JosephMaxwell            $candidates = $this->addressRe
3/28/20 JosephMaxwell                $this->searchCriteriaBuild
3/28/20 JosephMaxwell            );
3/28/20 JosephMaxwell
3/28/20 JosephMaxwell            $customerAddress = $candidates
```

```
commit 665aa61ff8224b488250688ab60c97434cb2f8e7
Author: JosephMaxwell
Date: 3/28/20, 6:15 PM

Finishing adding discounts to tests and downloads
```

There are also a number of merchants who are happy to receive push-back on their requirements (I'm sure we all agree these are the very best merchants to work with). Push-back comes from developers who understand what is needed. Work with your project manager (or account manager) to help the merchant understand the consequences of all approaches. For example, one approach might take a month to complete, versus your idea that will take a day to build (but will perhaps have a little less functionality).

Push-back shows thoughtful developers who desire to please beyond "just completing the requirements." Everyone loves people who go above and beyond.

Of course, you (likely) work for an agency so you must work within what they consider acceptable. I would wager a bet there is significant room within this window to overperform.

3. Get to work

Now that you've defined the task, you have all information necessary to start toward a solution.

You might need help

Unless this is an ultra-critical priority (i.e., the checkout is down), I suggest taking at least 15 minutes to gather information. Start into the *Take inventory* step to reproduce the problem (we'll talk more about this in Chapter 6). Write down your findings. You should have:

- The task reference (ticket number)
- The location where this is happening in the code
- Related error logs
- Unique aspects of the environment
- Potential solutions

Once you have collected these pieces, go to a teammate and ask for advice. If you are working with code written by another team member, start with that person.

If the website is down and you get assigned to this task (and you don't feel capable of solving it), immediately go to a senior developer. Ask for suggestions on how to resolve this problem. Do your very best to apply them in the context of Part 2 (debugging framework) and Part 3 (tools and strategies for solving problems) of this book.

If this problem is urgent enough, the senior developer might take the assignment. In this case, watch over her shoulder. You will learn from the process.

4. Consider asking for more time

If you are concerned that you are asking for help too often, ask your manager for her thoughts. She will appreciate your desire for improvement and concern about disturbing your other team members more than necessary.

When you hear words like “too difficult” or “unsolvable,” the counter word that should come to mind is “time.”

Does this scenario sound familiar? Your seniors avoid giving you challenges because they know you won’t be able to figure out the solution (this can happen even if you are perceived as hard-working and unstoppable). The result is that you only get tasks that are roughly within your skill level. Your company does this to bill enough to cover your salary + overhead + some profit.

Your opportunity to increase your challenge level is limited.

- Consider asking for challenges that are not bound by urgent constraints (i.e., a site is down and must be fixed immediately). This might be debugging a problem that occurs randomly, and the solution will take several weeks to effectively determine.
- Pair code with a senior developer. Watch how she fixes the problem *and take notes*. Taking notes is a way to communicate value and helps you to retain ideas in your memory.
- Take on challenging side/hobby projects. This could be fixing bugs in an open source project or building modules or plugins useful to your situation.

Spend your own time working on these challenges. Treat them as career-development opportunities. I am NOT advocating for your company to ask you to take this time. It **must be** your own initiative and for your own benefit. Companies regularly take advantage of developers with the idea "it is good for them," but in reality they are sucking any last drops of initiative and morale from their employees.

My experience suggests that the amount of extra time developers dedicate to their career (outside of the required time) is directly related to how quickly they ascend the ranks in their company.

But where do we get this time? Come on, we're busy, aren't we? I'm betting we all have extra time, somewhere. For example, might you be able to delay that new Netflix show?

Make sure to provide status updates at regular intervals (your manager will have a recommendation). This way, your manager will know the problem hasn't fallen through the cracks.

I guarantee you will learn more by spending your own time solving problems than you will on work time. As you become a master debugger and are able to use this skill during your workday, you will be a more valuable member of your team.

Don't give up

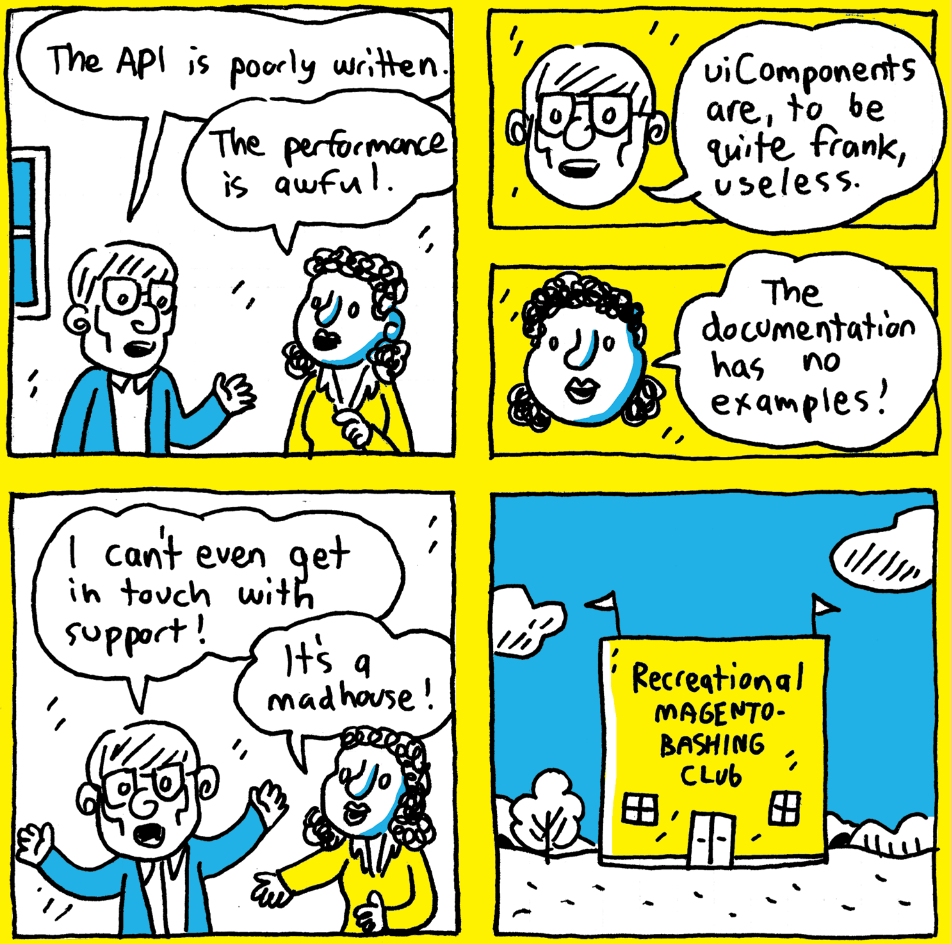
I began the chapter with an emphasis on mental fortitude. If you are a one-man show, you know this very well as you have no one to whom you can pass off a ticket.

The result?

You will gain experience with difficult concepts. You will gain more confidence in your work. You will make this learning your own. You will become passionate about your findings.

Chapter 2

METHODICALLY SOLVES PROBLEMS— WITH FLEXIBILITY.



The API is poorly written.
The performance is awful.
uiComponents are, to be quite frank, useless.
The documentation has no examples!
I can't even get in touch with support!
It's a madhouse!
Recreational MAGENTO-BASHING CLUB

A great developer uses an established toolset to quickly solve problems and strives to avoid tunnel vision at all costs.

This is my favorite characteristic and provides the foundational theory for Part 2: Effective and Fast Debugging, which you will get to soon.

You are both a detective *and* a builder. You must figure out problems and then build a solution to those problems.

One of the easiest differences to observe between junior and senior developers is how quickly they solve a problem. As we discussed in Chapter 1, it is my opinion that this difference is split into two categories:

1. **Experience in specific solutions to problems.** This comes with time. You fight through a challenge and learn the solution. Through the course of your career, you will learn thousands of solutions. Sadly, unless written down, these solutions fade with time (due to technology or the limitations of our memory).

2. **The framework to effectively solve problems.** This comes through knowledge, self-discipline, and practice.

Point #1 is something that only comes with time. If time and experience are the only factors that contribute to being a great developer, we have a problem, as we are impatient people who want to reach the status of a *great developer as soon as possible*!

On the other hand, Point #2 is something learned, and I will do my best to show you how it can be done.

A real-life example

In early 2020, one of our merchants moved to another agency because we were too expensive (I am a strong believer in the golden rule and did my best to facilitate a smooth migration).

This company was on Magento 2.3.3. One of the new agency's first tasks was to migrate to Magento 2.3.4. They tested it "thoroughly" on the staging site.

We have uptime monitoring on every site we maintain (and I hadn't taken the time to remove the monitoring for this particular merchant). I noticed the site went down around 10:00 a.m. Since all work had been transferred to the new agency, I placed myself on standby but did nothing further. The error messages shown on the site changed from one thing to another, but the site never came back up. They transact quite a few orders; the cost for this outage was escalating.

Sure enough, around 1:45 p.m., I got a call from the merchant, asking for help.

Here is what they conveyed:

> There is an infinite loop issue during checkout and there seem to be some other reports of this issue. The errors are below:

```
[2020-03-05 15:50:24] main.CRITICAL: Report ID: webapi-5e611fbfe4638;
Message: Infinite loop detected, review the trace for the looping
path {"exception":"[object] (Exception(code: 0): Report ID: webapi-
5e611fbfe4638; Message: Infinite loop detected, review the trace for
the looping path at /srv/releases/build-420961701/vendor/magento/
framework/Webapi/ErrorProcessor.php:208, LogicException(code: 0):
Infinite loop detected, review the trace for the looping path at /srv/
releases/build-420961701/vendor/magento/module-checkout/Model/Session.
php:241)"} []
```

Being flexible means you will solve problems sooner. You will recognize when you are hitting a dead end. You will become very effective in making your clients happy.

They already located the error message, so that was very nice and saved me time.

The checkout session's code was simple:

```
public function getQuote()
{
    $this->_eventManager->dispatch('custom_quote_process', ['checkout_
session' => $this]);

    if ($this->_quote === null) {
        if ($this->isLoading) {
            throw new \LogicException("Infinite loop detected, review
the trace for the looping path");
        }
        $this->isLoading = true;

        // …

        $quote->collectTotals();

        // …

        $this->isLoading = false;
    }
    // …
}
```

(I abbreviated the code so you can immediately identify where the error is happening.)

As with every problem, I determined to solve it quickly and methodically. It's go time! I kicked into high gear.

The first thing I tried was replicating this problem in my local development environment. That might have worked, given more time, but *I didn't have time—I was dealing with a critical emergency*. Instead of trying and trying to get a local replication, I moved on to something else. Try, fail, then try *something else.*

I was lucky the production environment was not a read-only file system. Making changes to files on production is normally a very bad idea, but given the extenuating circumstances, I had little to lose.

There are 61 lines of code between each `isLoading` statement. The easy hypothesis is the parent method, `getQuote`, is being called *a second time*, from code that is called within this method. In other words, something within these 61 lines is calling something else that ends up calling the checkout session's `getQuote` method. We have an infinite loop, and Magento added this error to prevent the loop from going rogue.

What code could be causing this problem? There could be a plugin (middleware) that initializes the infinite loop, but unlikely. We start with the most obvious and work to the least obvious.

This was the next suspect:

```
$quote->collectTotals();
```

My next thought was, "For crying out loud, how am I to find which total collector is triggering this?" I remembered a couple of things (my probability estimates included):

- 99% probability that core Magento will not cause this problem.
- 70%+ probability vendor-installed, quality third-party modules are not

causing the problem (these are installed with Composer and placed in the `vendor` directory).

The most obvious place to look then would be the custom code directories.

What would it take to drop a logger in the checkout session so I could trap the stack trace on this exception (using `$e->getTraceAsString()` – one of my favorite methods in PHP)? As a reminder, I'm walking through this process to show you how I troubleshoot a situation. The specifics are less important than the approach I am taking.

I wrapped the `throw new LogicException` in a try/catch statement and logged the error.

```
try {
    throw new \LogicException("Infinite loop detected, review the trace
for the looping path");
} catch (\Exception $ex) {
    \Magento\Framework\App\ObjectManager::getInstance()
        ->get(\Psr\Log\LoggerInterface::class)->critical('Infinite loop
stack trace coming:', [
            ‘trace’ => $ex->getTraceAsString()
        ]);

}
```

(See Chapters 8 and 9 for how to best use stack traces.)

For read-only environments, I could have created a patch and deployed the update to production. Once you get the logs, you need to make your fix and do another deploy. Read-only environments take longer to troubleshoot.

Sure enough, I found the problem in the stack trace (see Chapter 8 to understand what this is). A poorly-written custom quote total collector loaded the quote from the session a second time. The only reason this class tried to load the quote session was it needed the customer ID. When this class loaded the quote session to obtain the customer ID, an infinite loop ensued.

We would be tempted to ask “why would you load the quote from the checkout session just for the customer ID?” Because ... it is an easy way to obtain the ID! Just about anyone can make this type of mistake – even a senior developer.

My immediate fix became adding a `try/catch` clause to this custom quote total collector and preventing the total collector from running if there was an infinite loop.

The longer-term fix would be to consider other ways to obtain the customer ID. You could load in the customer session. You could load from the database, but that might not always be fresh. We could probably come up with five other resolutions.

With the problem fixed, I wrote the merchant and let him know how I resolved the problem. I gave pointers he could pass along to the other agency’s developers. **I felt like a genuine hero.** I really did. This fix cost their company about 30 minutes of my time – a bill of less than $100. The new agency had been trying for hours to resolve the problem. Such a small bill should have been the deal of the century.

I made a big mistake with *how* I communicated my solution to the problem. I only responded to my point of contact at this client. I didn't copy anyone else, even though I often talked with the client's CEO.

A few months later, I heard from their CEO. He told me the new agency said we caused the problem, but *they* fixed it. I was thinking, "What!?" I did my best to set the record straight, but there was only so much I could do at that point.

I would have done much better to copy anyone and everyone and share the good news.

How this is lived out

It is my strong belief that my framework with which I solve problems played a far more significant role than my experience as a developer (I had never seen this problem before).

In fact, this is the framework we will discuss in Part 2: Effective and Fast Debugging. You will learn the three steps of the TAD process and how to approach these problems. You will become very effective in making your clients happy.

Being flexible means you will solve problems sooner. You will recognize when you are hitting a dead end. You will become very effective in making your clients happy.

Don’t you love it when the merchant gives you information about what the problem is *and how to fix it*? After all, aren’t we developers the professionals and not the merchant? Aren’t we supposed to know our ecommerce platform inside and out? While it might be our tendency to fully reject ideas from our clients, I find value in giving some credence to these ideas. Merchants know their website better than we do. They use it every day. They hear from their customers regularly. Presuming we work for an agency, we might work on many websites and will not have the client’s level of familiarity.

Take the time to listen. It might be a completely invalid hypothesis. But at least this shows you care about the merchant.

Chapter 3

CONSTANTLY LEARNING NEW IDEAS.

Y'know, there are two IMMENSELY VALUABLE tools I keep in my development toolbox at all times.
And here they are in all their grandeur!
There's nothing there.
Experience and a problem-solving framework — the two intangible tools of priceless value!
So now can I have my box back?

A great developer is always on the lookout for new concepts and features.

Throughout my career – part of it spent teaching – I've been consistently amazed with how seldom new features are used. My hypothesis is that people develop habits, and it is their habit not to step outside of those habits. What compounds this is how few companies encourage their developers to use new features or ideas – some actively discourage it! I heard of a merge request being rejected because the author wanted to use an API call from the frontend, which is easy to implement and has less processing overhead. Instead, the company preferred that the developer use an old technique that is slow, more prone to errors, and provides less-structured data.

PHP 7 introduced the null coalesce operator (`??`), the spaceship operator (`<=>`), and typed arguments. When was the last time you used any of these features? They were made available in December 2015.

PHP 8 was released at the end of 2020. Have you read about (and are patiently waiting to use) the `nullsafe` operator? Maybe named arguments?

We can zoom in on your preferred platform. Mine is Magento, so I will use that as an example.

What features were released in Magento 2.3? Magento 2.4? I love asking these questions in interviews because it is *an indicator* as to how much the candidate is *personally invested* in a platform. A candidate doesn't need to be an expert, but at least showing some interest makes a big difference.

Do you have a solid understanding of SQL? How does your platform build SQL queries? How does SQL injection happen?

Yours could be anything (now that PWA is a part of Magento, the toolset is greatly expanded to include the React family of components). WooCommerce. Shopify. BigCommerce. Laravel. Symfony. Etc. These platforms undergo rapid improvements.

When was the last time you wrote code using a new feature? When did you most recently browse core code to seek inspiration for the project you are building? How recently have you taken five minutes to read your platform's documentation?

See our website at swiftotter.com/artofdebugging for additional resources on staying up-to-date in your industry.

How much do you know about the server environment in which your application is running? Talk to your devops person and ask for some resources to help you better understand this system.

Lee Saferite said this type of knowledge can help resolve some of the most challenging problems. What if you have cron jobs (scheduled tasks) that randomly stop? It could be because PHP is running out of memory for this task. It could be the disk is full. It could even be another process on this system exceeds the memory limit and the OOM killer (out of memory) chose this cron process for death and destruction. Even rudimentary knowledge of Unix-based systems will go a long way.

A real-life example

Other than incorporating new PHP 7 (and now, PHP 8) features into our work, I want to share one of my favorite and significant changes.

In 2016, Adam Watham wrote a brilliant book titled *Refactoring to Collections* (if you don't have it, I encourage you to get it) that I purchased and read. Our local PHP user group had extensive discussions about these subjects.

The general idea was to avoid using `foreach` loops whenever possible. Instead, use functions or methods that are designed for this application, like `array_reduce`, `array_map`, or their counterparts included in a framework, such as Laravel.

This changed how I approached code. One of the PHP user group's members wrote the Haystack library which I used in a project called Driver[1]. During the following few months, my usage of `foreach` dropped dramatically. It was then I learned that these methods are not as performant as `foreach`, so my usage became balanced and targeted. I started thinking "in what areas will this idea be best?" To this day, I regularly use both `foreach` and the `map/reduce` methods.

Even though my current learnings have been geared mostly toward marketing and business development, I still work to learn new ideas, on my own time. Here are two examples in the last few months:

- Did a full deep-dive of how Magento and ElasticSearch works.
- Made using RabbitMQ a part of my asynchronous workflow (very, very few developers use this tool).

1 https://github.com/swiftotter/driver

PHP Jeopardy

A few years ago, I attended a PHP conference in St. Louis in the heart of the United States.

Near the end, the organizers hosted a game: PHP Jeopardy. Contestants pick a category. The harder the category, the more "money" you make. A clue is given for the chosen category. The first contestant to hit a button, after hearing the clue, answers the question in the form of a question.

Three developers with a Magento background participated in this competition. These people were senior developers (one of them is now a COO of a large Magento agency). Out of 10 or so players, these three performed the worst. I was blown away. How could some of the best in our niche fail so spectacularly at the language we use every day?

I came to a realization that ecommerce developers *can fall into a unique sub-category*. We end up using a *platform*. This platform is often massive with all of its intricacies. The result is we don't dig into the *language*.

How often do you use the `Iterator` classes in PHP? Or, the `array_map` or `array_reduce` functions? Do your classes enforce `strict_types`? Do you know the difference between `\Throwable` and `\Exception`?

Example: the Magento Web API

There are quite a few Magento features I enjoy using that have not seen great adoption in the community. I believe one of the biggest reasons is people don't want to change their way of thinking.

An example is the Web API. Magento includes a way to configure anonymous REST API access. Many Magento developers don't use this to fetch information from Magento after the page has loaded. Instead, they opt for controllers: the same mechanism used for rendering a page. This latter route involves more code AND more overhead.

How this is lived out

- Have an open mind. Be willing to be challenged. Neither you nor I are the best developers in the world. We both have many areas where we can grow. We also have much knowledge to share.

  One of my favorite experiences was helping write Magento certifications. I always felt like my intelligence and experience were dwarfed by the other brains who helped to write the tests. I grew as I listened to them "argue it out" (all in good nature).

  In other words, listen and allow your mind to be changed.

- Be humble. I suppose this is somewhat similar to the previous bullet point. Are you willing to be challenged? What would you think if someone gave you a better way to do something? Look back over the past year: *when you have been challenged*, what was your response – defensive or graciously accepting? Even if you think the idea is stupid, or comes from someone who has clearly inferior knowledge, there might be a small amount of merit from which we can learn.

- Could it be true that 95%+ of what we read has no impact on our lives? It's possible. Make the opposite true and change your habits through what you read. Even practicing 5% of what you read will change your career.

- When you look at the documentation for your platform, take a few extra minutes to browse the related articles. You will likely learn something new or a memory will be triggered of something you already knew.

- Understand the big picture of your framework/application and how the two fit together. Many applications are huge monoliths. You will be faster at choosing a solution to a particular problem.

- Follow smart people on Twitter/YouTube (a shameless plug here for the SwiftOtter channel). Remember people still write blogs, and you can follow them on Feedly. Take one suggestion a week and implement it into your code.

- Write down what you learn. See Chapter 10 for a conversation about a personal solutions log.

- Put pressure on your account manager to sell upgrades to new versions (of course, you want to give the framework/language plenty of time to stabilize; don't upgrade until at least one patch version exists, and regardless, always evaluate the cost + effect to applying the updates). By getting onto new versions, you often gain speed and development efficiencies. When most of your merchants are on this new version, you will really notice the stragglers.

  A word of caution. Bear in mind this has the potential to cause problems: You develop habits with the new features and forget the old website is not yet on this version.

- Learn a new thing every day. Even tiny enhancements such as a new IDE shortcut or a new bash command count toward this. You could pick up a new library or a new language. Have you considered how functional programming would change the way you approach coding?

- Get involved in a community. For years, my brother and I drove 45 minutes (each way) to Kansas City to participate in the PHP user group. I spoke at a few meetings. It was a tremendous experience, and I learned much. But, alas, COVID changed this, too.

One benefit of making continual investments in your domain knowledge is that you will do better in interviews. Some of the easiest questions to ask in an interview are related to new features of a given language, platform, or framework. If you can't answer these questions, it reflects poorly on you.

A side note: keyboard shortcuts

What do keyboard shortcuts have to do with debugging? The faster you are at getting things done, the faster you will be able to solve problems. If a website is down, you will get it up sooner. If you are working through a task, you might shave off a few minutes, which will allow you to get onto that new, cool ticket even sooner.

You wouldn't believe how many times I have observed people operating their keyboard: one finger, searching for the key, smashing that key and ... repeat (also called the hunt-and-peck method). It's a good thing we developers aren't like that! Can you imagine how long it would take to write a few lines of code?

Ah, but I think we are still guilty of this transgression, in a different way. When was the last time you learned a new keyboard shortcut? When did you last copy code by going to the menu, clicking Edit, rolling your mouse's cursor down to highlight Paste and clicking it? Or, might you use the right-click context menu to copy code? Do you know how long that takes? Why not use Cmd/Ctrl+C? It's not that hard and a heck of a lot faster.

Have you studied keyboard shortcuts in Bash or Vim? What about your development environment, like PhpStorm or VS Code? When you use the debugger, are you clicking the buttons to advance to the next line or the keyboard? Cheat sheets are readily available for all of the above.

I might be missing something, but I have a hard time imagining how the mouse can ever be faster than the keyboard. Think about it this way: your wrist is anchored to a solid object and your fingers can hit specific keys through muscle memory (isn't this how we type?). There is no such concept with the mouse.

Years ago (hey, I was young!), I argued with someone a few years older about why I loved Windows and hated Linux (I even pulled the argument that Linux seemed demonic considering they use the term "daemon" and all terminal windows were black). I felt righteous in that I saw the light and was trying to proselytize him to the way of truth, i.e., Microsoft.

This friend made a wise comment about how in Linux you just think things and they happen – through commands and keyboard shortcuts. As I matured in my familiarity with Unix-based systems, I have come to agree.

As a rule of thumb, if you find yourself doing something with your mouse at least once a day, try learning how to do that action with a keyboard shortcut. Before long, people will be blown away when they see you working.

Don't forget productivity shortcuts, too. These enhance the speed at which you develop your application. For example, use the bookmarks feature to jump between commonly-used places in your code. Install the module that provides additional insight for the framework or platform you use. These will make your IDE smarter and more aware of what is being written.

Chapter 4

COMMUNICATES EARLY AND OFTEN.

3 A.M.
Text from client: MY SITE IS DOWN
Missed call and voicemail from client: MY SITE IS DOWN!
MY SITE IS DOWN!

A great developer quickly reports important findings.

You know the drill. A merchant emails you and says, “MY SITE IS DOWN.” You put on your firefighter helmet and run to the fire. You are sorting out what happened and are making progress ... And then another email flies in, “DID YOU GET MY LAST EMAIL?”

I have been there and done that.

Look, from my perspective, I’m just trying to get the site back online and these constant emails (or Slack pings from the account manager) are a distraction. It’s frustrating, maybe even borderline maddening.

Let’s flip this around to the merchant’s perspective. They sent an urgent email and haven’t heard back. They are very concerned because they can’t fix the problem themselves and are relying on you to solve it. They follow up, hoping to hear *something.* Could you be out for the day? Could you be in a three-hour meeting? *Would you please at least let me know that you’ve read my email?!*

When you start a task, acknowledge it – especially the important ones. Communicate that you are working on it. This could be as simple as marking a ticket In Progress. It could be sending an email response stating you have received the request for assistance and are investigating it.

Because I also function as an account manager, I am the one to receive these emails. My goal is to have a response back as soon as I read the email. I write something like:

> I’m on it and will have an update to you in an hour.

Nothing fancy or descriptive, and it takes less than 30 seconds. I simply acknowledge I have seen this and they are in good hands.

That's a nice start, but, as developers, you are the one to resolve this problem. Provide frequent updates as you make progress or find clues to the resolution. Communicate directly with the client, or ping your project manager (whichever complies with your company's policies).

As we will discuss in Part 2 of this book, you must eliminate your tunnel vision. Tunnel vision is when we are hyper-focused on one thing and lose sight of the big picture (tunnel vision can be life-threatening for those in the military or law enforcement). Humans regularly experience this in a time of heightened stress, such as when our lives are in danger.

I write this from a perspective of also being an account manager. Many developers don't have these duel(ing) roles. Depending on your agency's policies, you might be able to communicate with the merchant yourself. Or maybe you must update your account manager. Either way, quick and detailed reporting makes all the difference in the world.

A real-life example

Situations where the production site is down or experiencing a degradation of service are difficult. I wrap up each of the scenarios feeling mentally exhausted and needing a break.

In this story, the merchant's site was functioning well, up until....

2:12 p.m.

I get this email from Dave:

> We've had two customers call in in the last few minutes to say that the cart "froze" when they were trying to place an order. One was trying to submit the order, the other was trying to update their shipping address. In both cases the application just quit responding.

2:13 p.m.

I immediately respond:

> I'm looking into it and will let you know as I have more details.

(Note I should have established a window where they will hear back from me. For example, this could be, "You will hear back by 2:30.")

I notice that some Checkout API URLs display the home page. I size the scope of this problem and see that it affects *all API calls* (and the checkout makes significant use of the API). What would cause the API to break?

The first place I look is Varnish (a full-page caching system that facilitates very fast responses). It could be that something became misconfigured and Varnish is saving these API endpoints in its cache.

I exclude the `rest/` API endpoint URL from the Varnish configuration (this shouldn't be necessary, but it's worth a try). I don't think this is the solution. It takes the site down for a minute, but that's not a problem considering no one can place an order anyway.

2:16 p.m.

I send Dave a status:

> I just updated the Varnish configs and the site is refreshing...

I evaluate other approaches. The API is still returning the home page. I review the logs. I add logging and redeploy the container. Nothing is turning up.

2:30 p.m.

I bring in the hosting company because I am running out of ideas (we hadn't changed anything on the site for a week or two). Through this conversation, I hear they had just implemented some previously-stated-but-unenforced restrictions on the API. The net effect is when the API goes over the rate limits, the home page is shown. This brings the site to a grinding halt and was the result of this bizarre behavior.

The hosting company reverts the change.

2:42 p.m.

I send another update:

> The site is back up. I'm getting more information, but the [hosting company] said they had adjusted Nginx rate limits. Not sure why...

3:01 p.m.

I request reasoning for this update from the hosting company.

Then, I send Dave a final closure to this problem:

> Turns out, they have rarely enforced API rate limits. So they turned that on today... and it did not go well for a lot of people. They are revisiting this, and I don't think it will happen again.

Dave had a great attitude about this outage. Communicating early and often kept him loyal to our company. He knew he was in good hands as it was clear I owned this issue and wouldn't stop until the solution was 100% achieved. And now, I think I need some fresh air.

How this is lived out

Remember this: Those to whom you report (merchant, project manager, account manager, etc.) need regular updates on critical incidents. That's obvious. This takes discipline to break out of our tunnel vision. Another way to state this characteristic is **empathy.** If we are empathetic toward our client, we understand their happiness is invested in the problem we are working to resolve.

Communication regarding estimates

If you are in software development, you have to work with estimates. It's an unfortunate reality in this field of work. Another way to live this characteristic is to communicate back regularly about your progress in relation to the task's time estimate.

A merchant whom we onboarded recently shared their experience with the prior agency. Estimates were blown over and over again. Features were implemented poorly. The merchant received push-back, but it was only to keep the project inline with the poorly-researched discovery documents.

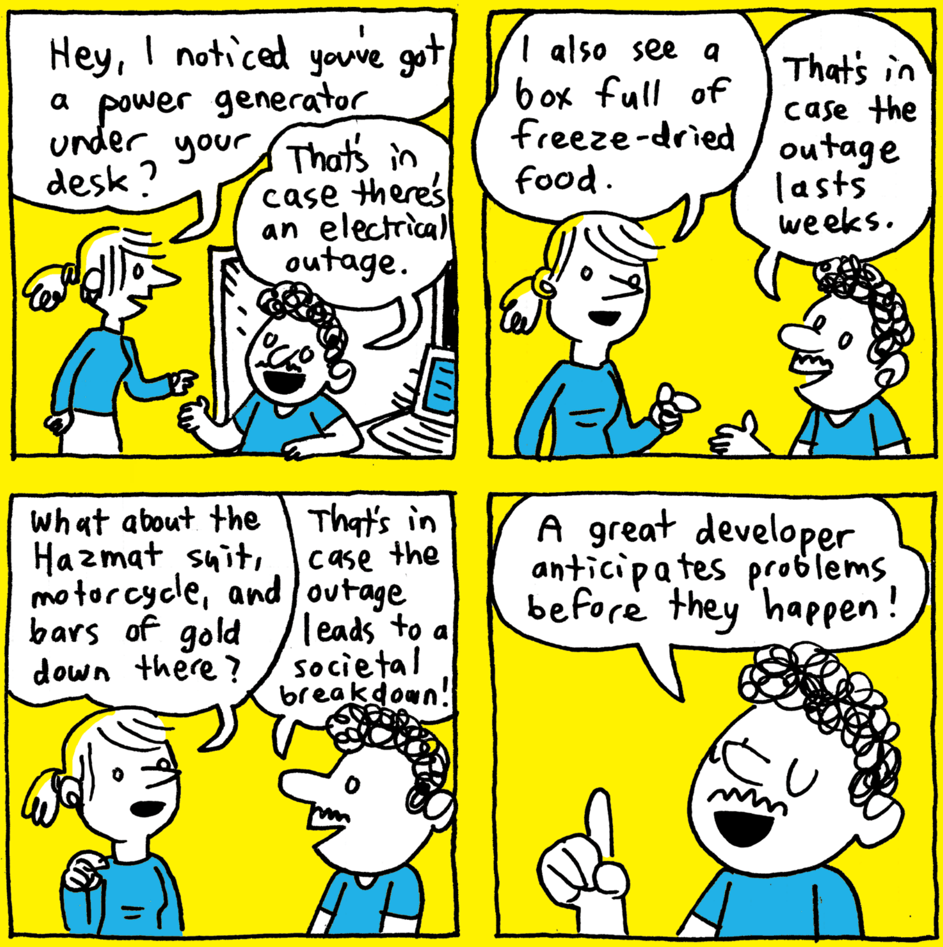
Blowing past the high of estimates is inevitable. However, we can do our best to reduce the impact as early as possible.

You might say this has nothing to do with your job. *I disagree.* The more you communicate relevant information, the more valuable you will become to all parties in your company. You are a rockstar developer in the making ... if you choose to be one.

Developers who communicate alleviate the fears of merchants – and become more respected and appreciated.

Chapter 5

BUILDS DEBUGGABLE APPLICATIONS.


Hey, I noticed you've got a power generator under your desk?
That's in case there's an electrical outage.
I also see a box full of freeze-dried food.
That's in case the outage lasts weeks.
What about the Hazmat suit, motorcycle, and bars of gold down there?
That's in case the outage leads to a societal breakdown!
A great developer anticipates problems before they happen!

A great developer anticipates problems and builds applications to make their future life easier.

My dad is an electrical engineer. Engineers design things in the real world. Through my growing up years, he imprinted on my mind the idea of how to approach building things. Much of this is boiled down into two key points: 1) everything is a tradeoff and 2) don't forget the **factor of safety**.

I remember when I built a deck for my dad. I used enough lumber that you could pack the deck full of people and park a car on it – all while still having extra weight rating to spare. Thanks to my dad's encouragement, I sacrificed speed of building the deck, and he sacrificed money for the additional lumber so we could achieve the tradeoff of safety and durability. The factor of safety is a derivative of tradeoffs and is an especially critical consideration when you are talking about human life.

These ideas are easily translated from building real objects into building software applications.

Everything is a tradeoff. We might also say everything requires balance. We could build an application so error-proof and tested that it never sees the light of day ("perfect is the enemy of done"). You will be constantly making decisions, but I find it is always best to take a couple minutes here and there to ensure you are building a stable application.

The factor of safety is critical. Our focus is on maintaining uptime for our clients. Because this isn't life-safety, we do have some flexibility, provided the client is on board.

Even in the most well-designed system, debugging is inevitable. You might pour over your code to ensure every semicolon is in the right place or every line is properly indented. Other developers might even sign off on your code through a pull request. Best of all, you might write unit/integration/functional tests for critical sections of your application. *And after all that, you will still have bugs.*

I argue that a characteristic of a great developer is looking into the future, knowing you will be back working on this code at some point. This is less about experience and intelligence than it is about eradicating laziness. We will explore ways in which you can help your future self.

Have you built a complex piece of code only to come back six months later and find you remember very little about your reasoning for this particular methodology? Remember, the time when you write this code is your opportunity to mitigate this frustration down the road. It takes discipline to think of your future self when writing code—but the payback is quite nice.

Defensive programming, "The factor of safety"

When I was growing up, a neighbor moved in a few doors down. They brought two furry creatures with them. The lighter dog looked like it had been rolling in decomposing dog turds and mud puddles. One of its eyes was white, the other was dark. I couldn't put my finger on what was wrong, but I got a "crazy" feeling just looking into this dog's eyes. The other dog was similar in looks but opposite in color.

We nicknamed the dogs Seek and Destroy, respectively (but *not* respectfully). The former dog was seeking, ahem, barking – all day and all night. I kid you not this dog damaged its vocal cords. Neighbors *hated* the beast. It was incredibly mean, too. Destroy was more tame, but it had a pastime of chewing anything and everything (Styrofoam coolers were like a birthday present).

The one lesson I can apply from these canines is this: We must be laser-focused on seeking and destroying problems *before* they cause trouble. This is the idea behind "defensive programming."

According to Wikipedia, "Defensive programming is a form of defensive design to ensure the continuing function of a piece of software under unforeseen circumstances." In other words, while we write code, we should be thinking, "What can go wrong with this line of code?"

As with anything, we must utilize careful thought to maintain balance. We can write code with no thought of possible consequences, or we can bloat our code with catching anything and everything – a significant portion being unnecessary.

Let's look at two places in which we can apply defensive programming. Here is the subroutine:

```php
public function getFirstItem(array $input)
{
    return $input[0];
}
```

This code is fetching the first element in the provided array. Have we considered what happens if `[0]` is not present (this is an empty array)? We will get an error – and the execution of this program will crash.

Let's improve this slightly:

```
public function getFirstItem(array $input)
{
    if (!isset($input[0]) {
        throw new NoSuchEntityException('First item not found.');
    }

    return $input[0];
}
```

We now have some feedback through the form of an exception. Let's investigate how to call this method:

```
$this->getFirstItem($arrayToCheck);
```

If we want to be ultra defensive, we could catch ALL errors, using PHP's `\Throwable` interface, like this:

```
try {
    $this->getFirstItem($arrayToCheck);
} catch (\Throwable $exception) {
    // … do something here ...
}
```

This squashes all possible exceptions and is very defensive, right? I suggest that in most every instance catching all exceptions is bad practice – unless you are doing something with it (e.g., logging or notifications). PHP's `\Throwable` class catches everything, including execution errors. You can get strange/dangerous results.

Applying defensive practices would say that an exception could be thrown – a `NoSuchEntityException`. We need to catch and properly handle this specific exception. Here is how I would do it:

```
try {
    $this->getFirstItem($arrayToCheck);
} catch (NoSuchEntityException $exception) {
    $this->logger->debug('Element 0 missing from array.', [
        'trace' => $exception->getTraceAsString()
    ]);
}
```

This code surgically targets the exception and logs to an expected place (using a PSR-4 compatible `LoggerInterface`). All other exceptions are thrown and may stop execution of the program. This is where having an application monitor or log analyzer, such as Sentry.io, is very helpful.

If we have no need to track where this exception is thrown, we can silence it by removing code executed in the `catch` clause.

As you write code, ask yourself, "What happens if this breaks?" Your framework might have good error handling for REST API requests but not for the command line. This is one way in which knowledge of your framework is crucial.

You might embed exceptions into your code, but what's the endgame for this exception? Will it be logged (are you logging it with plenty of extra detail)? If it's urgent enough, do you have a mechanism for being notified when this breaks? Will it be silenced? Will only the frontend user see it?

If you don't have an answer, *go find it*.

Don't trust any user input

Never, ever, ever accept arguments from a web request and pass them into your application *until* they are properly sanitized.

Every framework has methods to escape user input. Familiarize yourself with what they are.

PHP has functions beginning with `ctype` which allow you to understand the type of content that is passed in. Use them (it will save you from problems associated with blind type-casting).

The following are some key aspects (note the examples are from crude PHP; your framework likely has a more elegant approach):

- Cast numbers to an appropriate numerical type to ensure they are a number.

```
$quantity = $_POST['quantity']; // bad
$quantity = max(1, (int)$_POST['quantity']); // good, this converts
the type to a number, then ensures the quantity is at least "1"
```

- Sanitize text going into and coming out of the database. Use your framework's ORM (object-relational-mapping) system. Building SQL queries using quotes and strings can be very dangerous as they are an easy source for SQL injection attacks.

- Don't trust any POST or GET user input (I have seen hundreds of examples of HTTP request abuse). These inputs are particularly dangerous.

```
// BAD
$reviewText = $_POST['review_text'];
echo $reviewText;

// GOOD
$reviewText = htmlspecialchars($_POST['review_text'], ENT_QUOTES,
'UTF-8');
echo $reviewText;
```

- Use your framework's toolset for constructing URLs. If you render HTTP request data directly into a URL (and don't specify a domain name), you are opening yourself up for a request redirection attack.

```
<!-- BAD: -->
<a href="<?= "/catalog/product/view/id/" . $_GET['id'] ?>">

<!-- GOOD: -->
<a href="<?= "https://mycompany.com/catalog/product/view/id/" .
urlencode($_GET['id']) ?>">
```

- Never write a variable directly to the HTML output that comes from the database or a HTTP request (query string or POST data). These could be compromised and allow for cross-site scripting vulnerabilities (XSS).

- Enforce maximum length validation (you will have *that one instance* in which the length is exceeded).

Type hint everything

If you use a loosey-goosey language, like PHP, anything flies (pun intended). You never have to specify types – some developers simply don't. This is a mistake.

For example, what if you accidentally cast a decimal tax rate percentage to a whole number? The result is 0 or 1 – neither are valid tax rates. What if you assume a variable is always one type, but your co-worker has no idea of your intention?

While PHP has limited type-checking, you can use `docblock` annotations to specify additional details, such as "an array of a type."

I find forcing types on *everything* is effectively an additional level of "testing." Your code will simply break, allowing you to catch the problem early, in the Quality Assurance (QA) step – well before the customer sees it. You'll avoid strange, hard-to-debug calculation outputs that can cause problems in the future.

Every method should have a return type. What if I need to return a number OR a `null`? The answer might be found by running a conditional check *before* executing the method. This will ensure you can limit the number of cases in which you have to hybridize the return type.

What's your code style?

I come from a predominantly-Magento background, so I'm using an example in this domain. While Magento is not, in its entirety, a museum for "the best practices," most of the code base is at least decent.

There have been several occasions where I've interviewed a developer/agency for a merchant. I like to ask about coding standards and how they relate to the Magento codebase. These people assure me that their code quality is impeccable – and at least as good as Magento's standards. I then ask for some samples. You wouldn't believe the gremlins that I find in some cases. There are clear violations of best practices, guidelines, etc. If nothing else, indentation is terrible, cyclomatic complexity (how many indentations are used in a method, lower is better) is very high, and it is constructed poorly.

Study the code in the platform and the framework you use. Embrace best practices presented there and use them for yourself. Do be careful of legacy code that is still present in these systems and learn from the most recently-added areas. Don't stop with your platform. Learn from others (Laravel and Symfony are full of fantastic ideas). Match or exceed the code readability.

As a side note, I find smaller classes and methods easier to understand and debug. My preference is to have one `public` method per class (with any number of `private` methods). If you feel the need to have multiple `public` methods, so you can run better unit tests, you might take this as a sign to create a new class. I also work toward having less than 15 lines of code in any given method.

Build automated tests

"You should be embarrassed of the code you wrote last year—otherwise, you are not progressing as a developer." — Unknown

I am a firm believer that all critical code paths should be tested. Ideally, everything we write is tested – but that's not usually feasible given client budget and time constraints.

If we are honest with ourselves, we would agree that (most) developers do a lousy job of quality assurance. We love our QA teams ... until they find something wrong with our code. Then we hate QA. All kidding aside, I believe one of the reasons I hate testing is because there is no substance to clicking around and seeing if things work.

On the contrary, automated testing is built once, and it proves our system works through the course of time. When I am building critical systems (pretty much anything having to do with money), I write integration or unit tests at roughly the same time as the main logic. And, no, I don't write tests for *everything.*

My perception when learning about automated tests was that it seemed incredibly difficult: What are mocks or stubs? How do I get methods to return specific values without rebuilding the class?

Merchants won’t immediately see you as a great developer if you solve problems before they happen. They will appreciate your experience over time.

After building quite a few tests, I have found that it's not that difficult! In fact, the quality of my code is better, and I have fewer bugs. The bugs I do run into are usually centered on how the tested areas interact with the rest of the application.

But doesn't this take a significant amount of time? Yes and no. I used to spend time building my test cases through the browser. I would refresh the page or perform some actions so I could test my code again.

Now I invest most of that time in building an integration test that simulates and automates the testing process.

This is another area you can invest in and learn – and I bet you will never regret it.

If you are a junior developer, tests will seem scary. They were scary to me for the longest time. Consider studying this topic in your spare time. I guarantee the quality of your code will improve and you will experience less bugs.

One of the keys that unlocked the ability to write tests for me is small classes and small methods. It is very difficult and frustrating to test monster methods. Classes and methods that do one thing, and do it well, are often quite simple to test.

Enforce a policy for code reviews

Two heads do twice better.

Whether you are a junior or a senior developer, it is very important to have another set of eyes look at your code. As I share in these pages, I have made the dumbest mistakes, and they even slipped through the code review! Countless others are caught and prevented from arriving on production.

Version control makes this very easy – you create a pull request and someone else reviews your code and merges it.

We have caught an innumerable number of problems through these reviews. This means that the second pair of eyes prevents bugs from going to production. Even the best of developers make mistakes. We must have a good answer for why we did what we did and humility to accept the correction.

Include logging in your application

Isn't it fun to write code? Isn't it a pain to debug it?! Yes, for sure. The more you engage with the code you write and are willing to question it, the less time you will spend debugging!

One way you can question what you build is to ask yourself: "What is most likely to fail in the future?" I wish we could build bulletproof modules. It would be wonderful, but that's not how real life works. We are under the gun to get this project done *yesterday.*

The nice thing is that adding logging is fairly simple. To be clear, I'm not recommending you log *everything.* Instead, be prudent and use your best judgment.

Here are some areas that often fail:

- API requests. This could be invalid information that is returned, timeout errors, or server errors.

- Calculations. These are bound to experience weird errors. Someone's going to put in an input that was not anticipated. The more complex your calculator is, the more likely it is that a slightly-off value will generate massive changes.

- Changes to application workflow. If you need to substitute in a new class for an existing one, you have the potential to create great side effects. When taking these drastic steps, ensure you have logging in place to get the full picture *when* you need to fix something.

Remember logging is worthless unless you have quality messages and a good context:

```
BAD: code triggered SKU 12345

GOOD:
PriceIndexer encountered error on SKU 12345
Call stack: …
Store ID: 1
```

You are effectively anticipating potential errors before they happen. Do your best to immediately mitigate the obvious ones and add logging to shed light on those that could potentially be problematic.

Wrap up

As we wrap up Part 1, let's do a quick review of the five characteristics of a great developer:

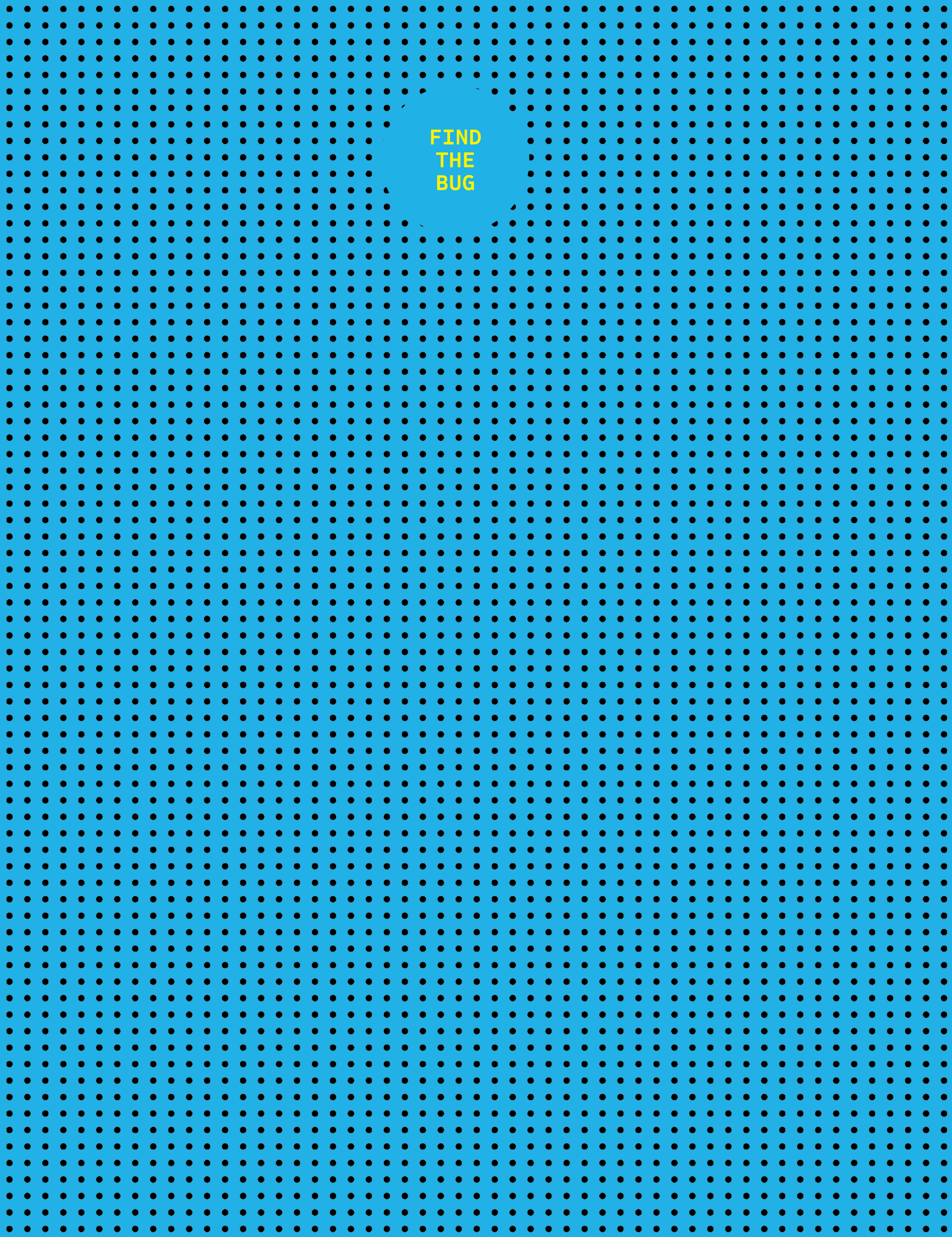
- A great developer *never gives up*. You will be a phenomenal asset to your company and to the merchants you work for as you grow this trait of persistence and determination. They know that they can always count on you to get the job done.

- A great developer methodically solves problems. By following the TAD framework which is discussed in the next part, you will solve problems *quickly*. Merchants have less down time when their problems are solved in record speed, and they make more.

- A great developer never stops learning. We all agree the "web is constantly changing," but the reality is it's easy to learn the basics and never progress. I'm working on a project where senior developers wrote the majority of the code ... and it's not good code. Take this in the right way, but I hope you are somewhat embarrassed of the code you wrote last year. This means you are progressing.

- A great developer communicates early and often. You recognize that other people's emotions are directly related to the work you are doing. Regular communication alleviates these concerns.

- A great developer foresees problems and builds debuggable applications. There are many problems that can be avoided given some thought. Do it and your life will be much easier.

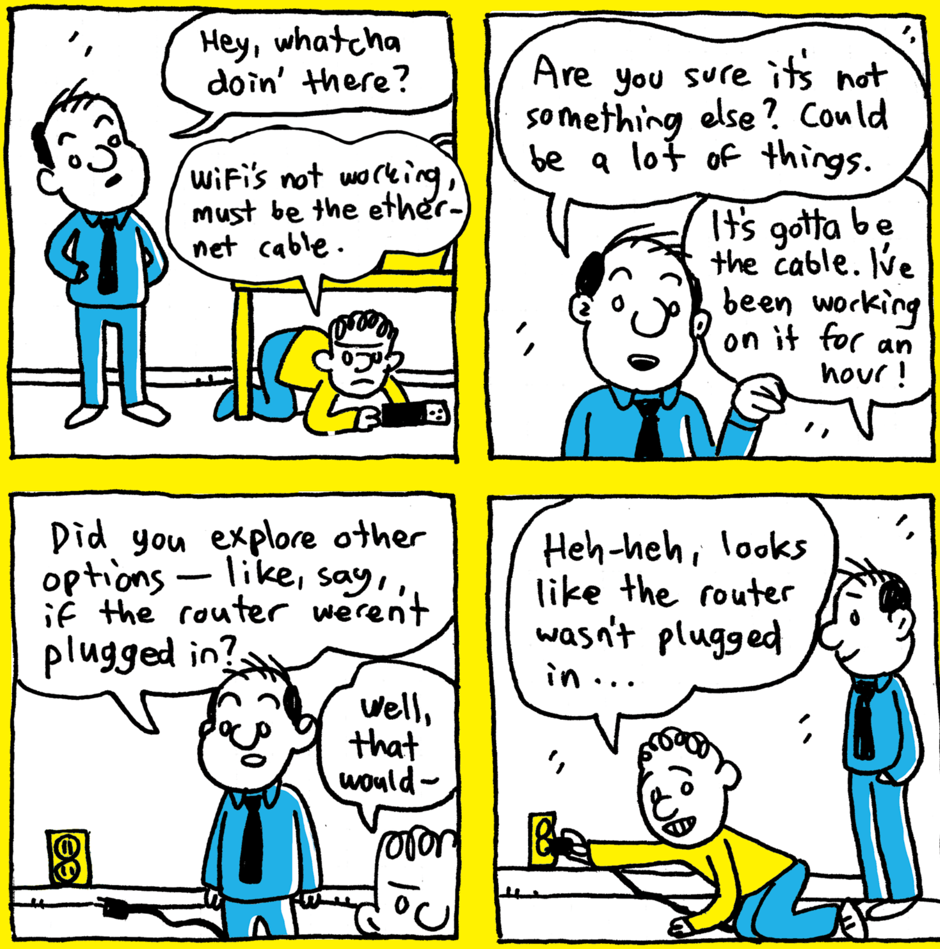

FIND
THE
BUG

Part 2

Effective and Fast Debugging

Chapter 6

THE DEBUGGING FRAMEWORK


Hey, whatcha doin' there?
WiFi's not working, must be the ethernet cable.
Are you sure it's not something else? Could be a lot of things.
It's gotta be the cable. I've been working on it for an hour!
Did you explore other options — like, say, if the router weren't plugged in?
Well, that would–
Heh-heh, looks like the router wasn't plugged in...

One of the biggest failings I regularly observe in junior developers is the inability to step outside of their tunnel vision. They lock into what they perceive to be the problem, and they work on building one solution for one problem. Instead of questioning their initial findings, they burn valuable hours continuing to modify their solution.

While I hope that junior developers look back to retrospectively observe this phenomenon, I rarely see this. Instead, the cycle perpetuates. An unfortunate (but natural) result of this is that junior developers struggle to receive progressively more difficult projects (see Chapter 1). Management cannot afford to delay a fix for a website outage.

In this section I present the framework that helps me to quickly solve difficult problems, and I believe any developer can apply this framework and become successful.

THE PERFECT PROBLEM REPORT
Communication goes a long way. The more information documented in an issue's description, the easier it will be for other developers to also work on the problem. Your issue descriptions should include:

- **A short description of the problem.**
- **Bullet points on how to reproduce the problem.**
- **A brief write-up of the expected results.**
- **And ... what is really happening.**

You might not have the details (yet) on how to reproduce the problem. That's okay. At least you will have a template you can populate as you harvest more information.

Overview of the TAD framework

- **T**ake inventory
- **A**ttempt a fix
- **D**o it again

Three steps? That seems simple – and it should be. This idea must become second nature and will revolutionize your ability to quickly solve problems. I believe it applies to fixing production outages as well as to solving the easiest bug ever.

I recently talked with Lee Saferite. He made this comment: "Debugging skills got me very far in my career." He made the point that *the ability to effectively troubleshoot a problem is a big difference between a junior and senior developer*. He said if you can be a junior developer who is self-motivated and knows how to debug problems, you are on a fast track to becoming a senior developer.

Note that by definition, this does not apply to architecting a new module. I find that debugging and architecture are very different paths, and there is little overlap. Senior developers are usually both excellent debuggers and architects. To keep the scope of this book narrow, I have focused on debugging.

Above all, don't give up. I have observed the temptation for developers to develop (no pun intended) a habit of escalating to their seniors. Senior developers solve the problem faster, and who knows, maybe better.

Think about this: Every problem you encounter is a step toward your leap to the next level. In fact, if you keep this up, I think one day you will find yourself the person who solves the most difficult problems at your company – and you will arrive at a most coveted position: **indispensable.**

You hear me make many references to tunnel vision particularly in junior developers. I think there is another corollary aspect: Could it be that junior developers are scared to try another approach? They feel as though they are wasting the time they spend looking for another hypothesis and solution.

But what if I am in a timebox? I am roughly halfway through my allotment, and I am not able to solve this problem (i.e., a client's checkout is down, and you don't know the problem). What should I do?

You have several options:

- **Explain your situation to a coworker or even speak out loud to a rubber duck (see Chapter 10).**
- **If you need to hand off the problem to a more senior developer, ask to code pair or observe as they work to resolve the problem. At the very least, review their notes and associated merge request (if applicable). Write this down in your personal solutions log (see Chapter 10).**

Take inventory

Target maximum time: 1 hour
Goal: a well-defined hypothesis of the problem

Definition of "hypothesis":

> a supposition or proposed explanation made on the basis of limited evidence as a starting point for further investigation.

We could also call this "Assess the Situation." As developers, we are almost always presented with *the symptoms.* Symptoms are:

- "The website is down!!!! PLEASE GET IT UP ASAP."
- "We are having random customers get an error when clicking the Place Order button in the checkout."
- "Online credit memos do not work."
- "The layout on this page is broken."

We venture into our troubleshooting with less information than we need to solve the problem. It is our job to fill in all missing blanks. This step is exclusively to take *some* time to gather information.

Symptoms are almost always NOT a hypothesis. Symptoms lead us to a hypothesis.

Don't become too attached to this particular diagnosis. Be ready to throw it away. You might have uncovered *a* problem, but more likely, you have not uncovered *the* problem. The earlier you are in your development career, the more likely it is that you have not found the right solution yet. Don't despair, just take your findings with a grain of salt.

If you haven't already, I suggest you read some Sherlock Holmes novels.

Holmes approaches situations and takes all pieces of the puzzle to come to a conclusion. On the other hand, his counterparts immediately identify who they think is the suspect. Holmes doesn't – until he is positive. He identifies patterns and uses these to reach his conclusion.

How do we do it?

- Let the merchant, your project manager – someone – know you have started working on this problem. Follow the protocol established by your company.

- Carefully read the problem report. I have found reading out loud can mean the difference between "getting it" and not.

- Convert symptoms into a helpful description. This is particularly important when the merchant sends five emails, all with additional information. Each email could be related to the problem – or not. For example, each email might contain a new URL, some with a different error (inventory problems are often this way). Make intelligent sense of what is going on. This will take some time, and you will feel pressure to avoid spending this time. But it's worth it in the long run.

- Start with the most simple place. I'd estimate this is the solution about 15% of the time. Is there a product's value that is improperly configured? What about the store configuration? I've spent hours troubleshooting problems only to come back and find the most basic and obvious solution. I felt like a pro, digging deep into the problem, only to feel like an idiot a short time later when the solution was so easy.

- If you are working on third-party code, such as a module, take a moment and see if there is an update for the module (you get bonus points if the module has release notes that specifically call out this problem.

- Ensure you are on the same page as the merchant. For example, a ticket that has "creating an invoice" could mean to you going into the admin, clicking into the order, and clicking the Invoice button to create a new invoice. This same verbiage could mean to the merchant going into the shipping software, creating a new label, and clicking the Create an Invoice checkbox. Your thought process means the invoice is created in the admin panel. Their thought process means the invoice is created via the API. There is a huge difference between the environment of each approach. Communication is key.

- Identify how you can replicate the problem. This should happen in your local environment. If you cannot replicate, please work through the steps outlined in Chapter 8 about troubleshooting "random" problems.

- This step is complete when you feel as though you have a solid idea as to what is wrong.

Example

We received this ticket:

> We cannot refund credit memos online (back to the credit card processor).

Somehow, I wound up troubleshooting this ticket. First, I took a look to identify where the credit memo button was configured in the code as this would give me an idea as to why it was missing. The logic to show the button was simple: If there is a `transaction_id` for this value, the Refund Online button is available. This problem changed from "the credit memo button is missing" to now the "transaction ID is missing." When the payment method captured, a transaction ID saved – but in this case, it wasn't being saved. See how this evolved? The initial time estimate increased, significantly.

I created a test order and submitted an invoice. However, the Transaction ID was present. Every invoice I created worked as expected.

An important concept to remember is to **challenge every assumption.** In my last paragraph, I used the word "created." Things don't always happen the way we think they should. Might there be other ways to create a credit memo? It turns out, there was. Keep reading to find out the end of this story.

My hypothesis of the problem is that "the Transaction ID is missing." I don't know why it's missing, but I'm running out of time, and I'd like to get more information by proceeding to the next step.

This example is perfect to show not everything will fit within the time guidelines I've shared. Hold those loosely but make sure you check in regularly (with yourself) to prevent tunnel vision.

Attempt a fix

Target maximum time: 2 hours
Goal: a solution for the problem

This is where you (finally) get to try your hand at a solution. Even if your hypothesis is not well-defined, attempting a fix will help you gather more information toward the solution.

The goal with this step is to build a solution that solves for your hypothesis (remember those old math days?). Don't bother with creating plugins or whatever your framework uses at this moment. You want to verify, as fast as possible, that you have arrived at the solution.

Watch for collateral damage. Just because something appears to be a "core bug" and you "fix it" does not mean that it is a core bug. There might be middleware causing the problem. *Your code* could be modifying core functionality. If the code is in a reputable third-party module, give it a 50/50 chance of being a bug. If the code is in core framework code, take a moment to search the framework's code repository to see if this is a known bug. If a bug is not documented, I suggest skepticism in defaulting blame to the core framework.

Now keep in mind you shouldn't hold yourself to a fix on this step. If you aren't making progress, don't hesitate to jump to the next step. Remember, the goal is to be agile in coming up with solutions.

How do we do it?

- Write or modify code. If you are in a local or staging environment, feel free to make changes to core code. Notate these changes so you can revert them at a later date.

- Test your work. What other code paths arrive here? Have you accounted for them?

I cannot stress enough how important it is to check in with yourself every hour or so. *Do not spend a significant amount of time on this step.* If you are a junior-level developer, cap your time at two to three hours. As you grow in experience, you will get a gut feeling that will guide you on how much time to invest.

Once you hit your time limit, jump to the next step. Free yourself from any guilt over not having achieved a solution. You are better off moving on than continuing to fight your way down this road.

Example

That Online Credit Memo problem that we discussed in *Take inventory* was a little tricky. I like observable problems. Because I was not able to replicate the missing Transaction ID through any steps to manually create the invoice, I came to the assumption that invoices are created through a different mechanism – likely automatic.

I went back to the merchant to double-check their process for fulfilling orders. It turns out, the shipment and invoice are created through ShippingEasy. I know there is no ShippingEasy module on the website, so it has to connect through the Magento API. If ShippingEasy can connect through the Magento API, so can I.

I configured Postman to trigger an API request to create an invoice. I was excited as I knew *this was the answer!* But alas, the Transaction ID was configured for this invoice. *Pffffft.*

So, how *else* could this be created? I used the API, right? There had to be some other way.

If the invoice was being created automatically, but not through code, it is certain that this is associated with a web request.

The solution for this *Attempt a fix* step (remember, we can have many rounds) is to add logging. I added this patch:

```
\Magento\Framework\App\ObjectManager::getInstance()
    ->get(\Psr\Log\LoggerInterface::class)->error($_SERVER['REQUEST_
URI']);
```

Yes, it looks horrid. If I see this code anywhere other than in an emergency debugging setup, I see a code smell that is worse than a bag of vomit. Or, milk that's gone *really, really* bad (not sure which is worse).

Here is the URL value that was logged:
https://[domain name]/rest/v1/invoices/12345/capture

I now had something to work with. The invoice was created from one API endpoint, but it was captured from another! Bingo, that was our solution.

I took this information to the next step and tried to replicate this locally. I achieved the local replication. Using the help of my debugger (see Chapter 9), I was able to easily solve the problem (and I do think this was a core bug, or at least a core oversight).

If I'm working on a local or staging site, I have no problem, whatsoever, with changing core code *temporarily*. Ben Marks gave solid advice never to touch the core. That's right 99% of the time. But if you need to figure something out, don't hesitate to make that change (but note this change so you can revert it) *and then build a plugin or middleware to permanently make the change.*

Do it again

Target maximum time: 15 minutes
Goal: recalibration and ready for taking an inventory

You have arrived at this step because you have not yet achieved a solution. That's okay. There is no judgment for not having an answer.

First round will often yield a short-term fix. You usually have to work through a second round to achieve a long-term fix.

This is your moment to assess where you are.
If you are stressed, take a walk. By taking a break and getting fresh air, you will clear your mind, which will help you to arrive at a solution faster.

If you feel you are out of answers, talk to someone. Talk through your scenario with others on your team. This is the "Rubber Duck" tool in Chapter 10. I can't tell you how many problems I've solved by simply repeating, out loud, to someone else, what is going on. To be clear, the other person fully comprehends the situation less than 50% of the time. They will do their best to throw out a solution, but don't count on their ideas being the savior.

This is your moment to communicate.
Add a comment to the work ticket (or send an email to the merchant, depending on your company's policy) and say, "I'm still working on this and haven't come to a solution."

One thing to consider is that you can use the "target maximum times" to estimate your time until a resolution. This is especially helpful if you feel as though you are close to a solution. For example, you might believe in the next troubleshooting cycle, you will have a fix (and you give yourself three hours for each cycle). Send an update like this: “I am hopeful that I will have a fix in the next three hours.”

This is your moment to document.
If you use a ticket-tracking system like Jira or BaseCamp (you should be!), take this as a moment to document your thought process. I also often use a text editor. IDEs like PhpStorm or Visual Studio Code can create “scratch files” for dumping your thoughts. However, placing this into the work ticket has the additional benefit of helping your fellow developers.

Copy in any queries you try (with a short explanatory note). Use bullet points to document what you try for each phase of the TAD framework.

The notes do not have to be pretty. This is information, and it is super beneficial. It helps avoid repeating troubleshooting steps.

Example

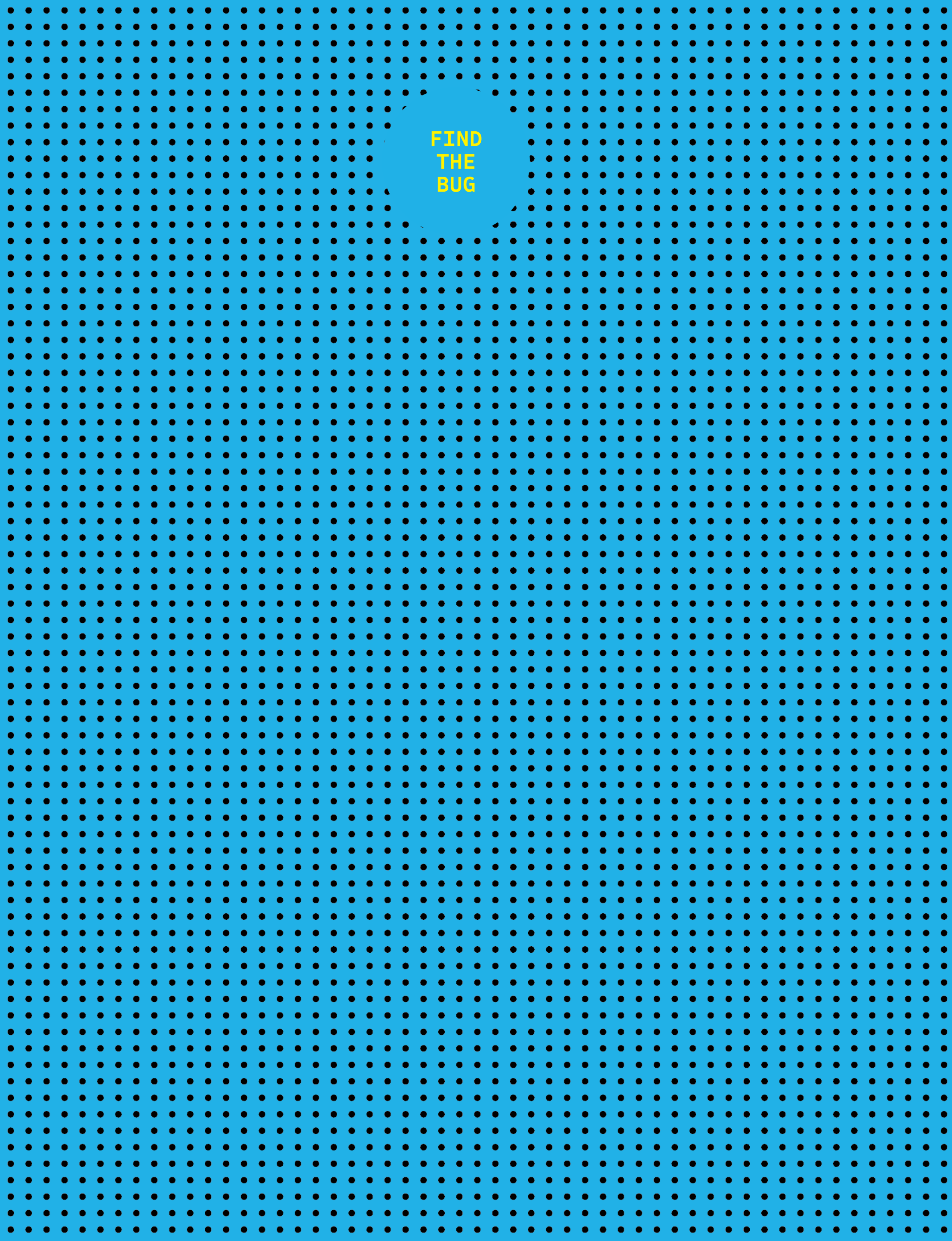
There are two tricky examples that come from both ends of the spectrum.

The first is the example I shared above on the client not being able to refund credit memos online. This was a waiting game. Results took time to see, so this *Do it again* step was embedded into the course of calendar days. The logged responses didn’t happen immediately, which means I could easily lose track of them. Losing track means more load on the Project Managers because they have to follow up on your status – and they don’t like to do that.

The second is where I am “under the gun” to quickly solve a problem. I don’t want to take a break. I want to keep pushing as I know the merchant is losing money every moment their site is down. If you have heard of the idiom “one step backward, two steps forward,” I believe it applies to this. Taking one step backward allows you to reassess the situation. Then you can push forward with more certainty.

Wrap up

Are you ready to tackle this new framework? I guarantee you will benefit from this and will significantly improve the speed in which you solve your problems. You will break out of tunnel vision. You will observe other possibilities and narrow down to the real problem.


FIND
THE
BUG

Part 3

Tools and Strategies for Debugging

Chapter 7

TYPES OF PROBLEMS TO SOLVE

Hey Bob, what's wrong? You don't look too happy!
I can't figure out why this data won't render right. It's killing me!
Well, I hate to say it, but you stressing over this is actually prevent-ing you finding a solution.
So the real problem is my stress?
Yup.
How STRESSFUL!

Overview

Yes, the theories we've discussed thus far are good concepts, and I'm guessing we would all agree we should follow them in one way or another. It is, of course, up to you to implement these concepts in your life and work.

But how do we actually solve problems? We must first categorize the problem we are facing. In building this section, I reviewed 114 recent problem reports from our merchants and split them into the following list categorized by their symptoms.

What I find especially fascinating is evaluating the amount of time tracked as an indicator to how difficult each category is. You will clearly see a trend.

1. **An error message is displayed upon taking a specific action**
   21% tickets
   18% of time spent

   Searching for the error message (unless directly passed from a third-party API) in the code base will yield a good idea as to what is happening. These usually point to a very specific place in the process of execution that is failing.

   I also put into this category any time an HTTP error code response is returned, such as the dreaded 500 error. Provided your web server is properly configured, you should be able to easily fetch logs for this type of error.

2. **Information is appearing, disappearing, or shown incorrectly**
   41% tickets
   42% of time spent

   This means something is wrong in the database or code. These problems usually happen as a result of a traceable change (code, database, or external dependency). The key is getting the pathway to where this problem occurs.

3. **Data persistence is broken**
   11% tickets
   13% of time spent

   This could be random updates that happen in the database. Or, worse yet, random deletions. Management (or your clients) will be asking why you would put such stupid functionality in your code. Worse yet, they have no idea why you can't figure out the solution to such an obvious bug! Doesn't an obvious bug = easy solution?!

4. **Visual issue (frontend only)**
   19% tickets
   9% of time spent

   This is when something is out of alignment, design is misapplied, or jank happens. You frontend developers will work on a proportionately higher number of these.

5. **An event is not being triggered as expected**
   4% tickets
   9% of time spent

This is one of the more difficult problems to troubleshoot. Obtaining evidence, in this case, means you understand the pieces of the puzzle that make this event trigger and what happens in this event.

6. **Performance problem**
   3% tickets
   7% of time spent

   The Time to First Byte (TTFB) is multiple seconds. JavaScript takes too long to initialize. You know every second costs your client money in the form of lost sales.

Keep in mind all of these symptoms represent an initial and first angle of attack. You must follow the TAD plan (see Chapter 6) and be agile to respond to new information you uncover as you troubleshoot this problem.

Work until you have a real solution

In the beginning of my ecommerce development career, I would often respond to problem tickets and tell them “it is fixed.” *The client would (almost) always come back with, “Why did this happen?”* This was made quite clear to me when I sat down with a merchant and asked for his feedback about our services. He said he would appreciate more information about how we solve the problems he reports. Hey remember, merchants are people too. It’s their website, and they deserve to know details.

One reason we might not want to give too many details would be if we know our patch is not a real solution. It could be a Band-Aid that works for a while only to have the issue return. I get it. We won’t always know if every solution is a long-term solution. But we can strive for it.

This is particularly evident with anything that cannot be immediately understood – and I'm looking at point #5: "An event is not being triggered as expected." These are the most difficult and irritating problems to solve (especially if we have new features in the pipeline to build and we just can't shake these old problems). We will talk more about this in a section dedicated to point #5.

Merchants love information and long-term solutions, even if it costs them more upfront. They hate issues that "reoccur" and having to spend money to fix those issues.

The next problem with quick fixes is they actually prevent long-term fixes. For example, if a problem is easily solved by deleting a row from the database, DON'T DO IT. Save that row's INSERT SQL in your work ticket. If you have the time, reproduce this situation locally. Then delete the row.

These symptoms are your forensics, and you must treat each carefully. You are a detective. Detectives *carefully* save any and all evidence.

I have delayed many long-term solutions because I went in and cleaned up data too quickly. Save the evidence as you will need it later.

Resist stress

It is my goal to be a fairly chill person – few things will get me riled up … except for a customer's website that is unable to transact. My adrenaline levels are directly proportional to a merchant's yearly revenue and the length of the outage. I can tell this because my normally *un*noticeable heartbeats turn into pounding palpitations.

When it comes to solving a difficult problem, can you identify your worst enemy? Yes, it's stress. Stress is an excellent way to cancel creativity. What would be your biggest asset for solving a difficult problem? Creativity.

As counterintuitive as it might seem, do everything you can to push back on those rising adrenaline levels. Temporarily mix up your environment. Get some coffee. Change your desk so you can work standing up. Maybe even a short stroll around the building (not long, of course!). Yes, the customer's site is down, but I guarantee you will be thinking about it; your creativity will be at its peak performance.

Please don't take this as me saying you *should* get up and leave when there is a difficult problem. What I'm saying is you must fight stress with everything you have ... so that you can win the battle.

An error message is displayed upon taking a specific action

I love error messages. They are the gateway to a solution. The solution can be difficult to find, but error messages give you a concrete starting place and context.

I lump 500 Internal Server Errors and related errors into this category because I can find related entries in the log files. Again, these also provide a starting place and context.

No value for table?

I notice a bazillion of these warnings in my log files.

```
[2020-12-19 08:00:11] main.WARNING: Field amount_estimated has no value
for table sync_activity_job [] []
```

We start by searching the codebase for the error. But there are no results for this. As I read the message, I see the table name is present: `sync_activity_job` and so is the field, `amount_estimated`. I remove these "bookends" and search for:

```
has no value for table
```

Bingo, I have results. The following extract is taken from a PHP class that parses XML content and then pushes this information into a database:

```
foreach ($fields as $field => $options) {
    try {
        $item[$field] = str_replace(
            ',',
            '&#044;',
            htmlspecialchars($record->filter($field)->text(), ENT_QUOTES)
        );

        if(empty($item[$field])){
            if (array_key_exists('is_require', $options)) {
                $this->_logger->critical(
                    sprintf('Required field %s value for table %s is not find',
$field, $table)
                );

                $progress->advance();
                $errors++;
                continue 2;
            }

            $this->_logger->warning(
                sprintf('Field %s has no value for table %s',$field, $table)
            );
            $warnings++;
            continue;
        }
    } catch (\Exception $ex) {
        // …
    }
}
```

The good news is, we now know **where** this is happening.

Our first task in *Take inventory* is to find the error. The second task is to understand **why** this is happening. We need more information. Why is the `$item[$field]` empty? It would only be empty if `$record->filter($field)->text()` returns a `null` value. But where does this come from? We must simulate the environment that calls this method.

Download the XML file. Set a breakpoint in this method. Trigger the code. You should quickly identify the problem. This was a data error. We then decide whether to fix the data or make this method better at handling errors.

Something (ridiculous) went wrong?

Error message when browsing a list of products:

```
Attention: Something went wrong.
```

This happened while modifying the filter of a Magento grid in the admin. Something caused a report no longer to load. Once the merchant receives this error, they can't use the grid.

The short-term fix was to delete out a row for the "bookmark" that was then saved with erroneous information. This is okay to get the merchant going but will not be the solution to their problem (very important distinction here).

These can be tricky to reproduce as they involve a delicate scenario with numerous steps that must be followed exactly. Here is how David (who works with me) solved the problem:

- David knew the table in which this bookmark is stored (`ui_bookmark`). This comes through knowledge of the platform.
- David copied down this value into his local environment and reproduced the problem. He reverse-engineered the JSON object to formulate a replication method for QA.
- David ran this on his machine and found there was a problem with a specific product (it didn't have a required SKU). He added this value to the database in production and solved the problem.

Entities only mean something when they have a cart ID.

Error message when placing order:

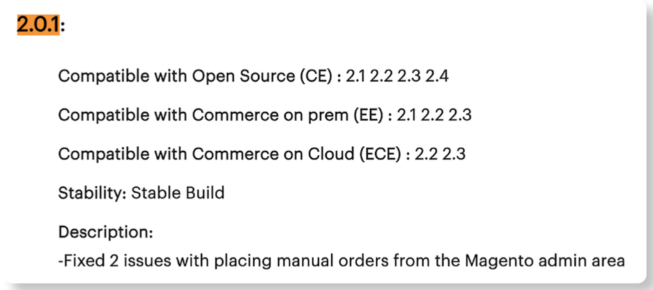
```
No such entity with cartId =
```

This one is painful. Yes, it's an error, but it's the quality of a genuine Windows XP message (wow, that actually was a "thing" years ago!).

David handled this problem, too. He loaded the production database down to his local environment and placed the order in his local environment. He stepped through the code (this is discussed, in-depth, in Chapters 8 and 9) and found that a cart was trying to be loaded ... but without an ID.

If the cart was without an ID, wouldn't we simply save it (to get the ID)? He started down this path, but then he had a moment of inspiration.

There was a third-party module we had recently updated. "Might there be any further updates to it?" He went and checked the change log for the module and found this entry:


2.0.1:

Compatible with Open Source (CE) : 2.1 2.2 2.3 2.4

Compatible with Commerce on prem (EE) : 2.1 2.2 2.3

Compatible with Commerce on Cloud (ECE) : 2.2 2.3

Stability: Stable Build

Description:
-Fixed 2 issues with placing manual orders from the Magento admin area

Yes, the problem was solved by updating the module. Many times, something obvious (like a missing ID) is not quite so obvious. Saving the cart object would have represented a quick-fix that did not solve the bigger problem.

But what if I have a generic error message?

Generic errors you find in log files are better than nothing, but they still aren't great. In PHP, you can update these messages by adding the `__LINE__` and `__FILE__` magic constants:

```
$messages = [
    new Phrase('Invalid Form Key. Please refresh the page.' . __FILE__
. __LINE__),
];
$exception = new InvalidRequestException($response, $messages);
```

Better yet, you can add the current stack trace to the log files.

Information is appearing, disappearing, or shown incorrectly

This is what I call "the gateway category." I estimate that in at least 50% of these cases, I uncover another problem. The initially reported problem is on the road to the underlying issue. As you read the examples below, you will see what I'm talking about. In these cases, Round 1 of our Debugging Framework is usually investigating the reported symptoms. Round 2 and later work to understand the root cause.

The first question I ask is, "Why is this data not rendering correctly?" Sometimes this is simply a bug in how the value is rendered. Other times, the wrong row is loaded from the database. Or it could be the wrong value is saved into the database.

Information disappears when we are missing important information.

The reason for rendering data incorrectly can be found in tiny missteps. This is a story that illustrates the importance of great care when writing code or debugging.

We built a custom calculation app for a client who designs safety systems for industrial purposes. Even in a system where we have solid testing, things still happen.

The number of parts chosen for the specified number of corners was not correct. This problem could have been a data issue for how the number of corners was handled or the calculation for the number of parts. The latter is normally the better suspect because it is more complex, and thus it is suspected (the opposite of Occam's razor).

I traced through the parts calculator and found it to be accurate. The next place to look is where the number of corners was calculated. Sure enough, I found I had written a `>` instead of a `>=`, which should have been there.

This is a classic case of incorrect information. And it can be costly.

Are you sure we have the right price?

Customer reported that an item is showing a different price on the PLP (product listing page) than on the PDP (product detail page).

(Round 1) *Take inventory*: What could cause prices to be different between the PLP and PDP? Knowledge of the platform tells us that PLP pricing is indexed ahead of time while PDP pricing is calculated on the fly. This gives us our first place to check: Let’s see what values are in the indexed tables. Ah, those are different, so we determine this must be because the indexer is not running or the indexed values are incorrect (former is more likely as the latter would be a core bug). Let’s proceed with our hypothesis that the indexer is not running.

(Round 1) *Attempt a fix*: We run the indexer. Great! Prices are now accurate. We could go back to the merchant and say, “The problem is resolved.” Or we could **and should** proceed to the Repeat step.

(Round 1) *Repeat*: We have identified the first part of the problem but in no way are we done. We can take this moment to update our account manager or merchant.

(Round 2) *Take inventory*: We now must answer the question, “Why does the indexer not run as it is supposed to?” We are investigating a totally different problem, and it would now fit into the “An event is not triggered” category.

(Round 2) *Attempt a fix*: We investigate the cron logs to understand why the price indexers are not running. We find the logs indicate missing runs. It turned out there were too many cron actions to be executed consecutively in a minute. We changed the configuration so that specific actions were run through the message queueing system – and the problem is now solved.

Potentially my most costly mistake ever

There are many days where I feel great about what I've built – not realizing there are sometimes long-term consequences to mistakes I didn't realize I was making.

One of these examples is especially painful. Remember that engineering calculator app we built (just mentioned above)? Part of this is a system to create a bill of materials (BOM). This BOM has a price that will be used for quotations.

I copied some code, tweaked it, had someone review it, and it was deployed to production:

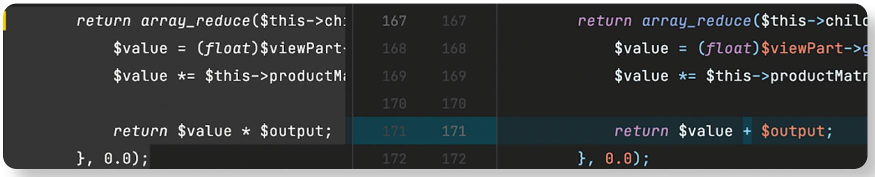
```
return array_reduce($this->children, function(float $output,
ViewPartInterface $viewPart) use ($type) {
    $value = (float)$viewPart->getQuantity() * $viewPart-
>getUnitPrice($type, true);
    $value *= $this->productMatrix->getPriceMultiplier($this-
>permutation->getProject());

    return $value * $output;
}, 0.0);
```

Several weeks later, I heard the pricing was inaccurate. Inaccurate pricing means that money is being lost and is probably one of the worst nightmares a developer can experience.

I dug into this problem and spotted it. Do you see the error? Yes, it's on line 6 where you see the `$value * $output`. The solution was easy. The fallout could have been catastrophic. Thankfully, it was spotted soon enough that the quotes were able to be updated and there were no further problems.

Here is a screenshot of my commit:



```
    return array_reduce($this->ch
        $value = (float)$viewPart
        $value *= $this->productM

        return $value * $output;
    }, 0.0);
```

```
    return array_reduce($this->child
        $value = (float)$viewPart->g
        $value *= $this->productMatr

        return $value + $output;
    }, 0.0);
```

I learned a couple of things:

- Every critical piece of functionality *must* have automated testing (unit and/or integration). Anything related to money (pricing, taxes) qualifies as critical. I am committed to this idea to this day.

- Our manual QA process did not catch the price problems. We updated their testing procedures to ensure the product that triggered this great error is tested.

This could have been way worse than it ended up being. A single-character difference can cost hundreds of thousands of dollars. **Take the time to ensure your code is correct.**

Data persistence is broken

“An entity doesn’t save” – that’s usually pretty easy to fix, and we resolve most of these in development. But some problems are more difficult to work through.

The easiest data persistence problem ever

Some columns are randomly truncated. A maximum column length issue is very easy to spot.

One of the hardest data persistence problems to spot

Incorrectly stored unicode characters. PHP’s `json_encode` function is notorious for not being able to understand or handle these characters. The result is `null`.

Another difficult issue

Here was an interesting problem. We were saving a value for which a MySQL relationship was present. The value was `null`, but the REST API cast the value to an `int`, which means the result was a `0` (`null` does not strictly equal `0` in PHP). The value in the `INSERT` query is `0` – and there is no corresponding row in the related table. We get an Integrity Constraint Violation error.

There are two angles to approach this. (1) We can modify the core REST API code to prevent the casting of this number. But we have potential fallout as we are not sure what other processes intrinsically rely on this casting capability. Or (2) we can modify the handling of the save operation for this model. We opted for the latter to prevent unintended consequences.

When something looks right, but it isn't

This can also be a gateway to another problem. And, this is why the debugging framework is so important.

For example, we were troubleshooting why a Point of Sale application did not mark an order as complete (Point of Sale is used in a brick-and-mortar establishment so a sales clerk can facilitate a transaction; this is especially helpful in that the order is synchronized with the ecommerce platform to maintain unity in the inventory system, you know, that "omnichannel" buzzword.). All orders purchased in-store should have both invoice and shipping records and be marked complete.

Take inventory: We identified there is code in the Point of Sale to mark the order as complete – but this code was somehow not being triggered. We could have built another trigger for the same code, or ...

We reviewed the PHP error logs (indicating a fatal error) and found this:

```
getExtensionAttributes expects an instance of \Magento\Sales\Api\
Data\OrderExtensionInterface, \Magestore\Webpos\Api\Data\Checkout\
OrderExtensionInterface returned.
```

The problem was that an event was being called with a different object than expected.

Attempt a fix: We added middleware to prevent this "different object" from proceeding through the system.

This is a good example of why you should enforce type-checking whenever possible. The above is an incompatibility between two third-party modules. If type enforcement was properly implemented, these errors would have been caught in development as opposed to sometime in the future (as we experienced).

Also, this can prevent costly rounding errors. For example, in PHP, `float` (with decimal point) and `int` (without a decimal point) are two different types. Careless casting of floats to integers removes everything after the decimal place (for example, `3.91` becomes `3`). This can lead to catastrophic consequences.

Visual issue (frontend only)

You backend folks can skip to the next section. This is not applicable to you unless … you work on the frontend. I'm guessing you do from time-to-time.

Why can't I check the checkbox?

Customer informs us the contact form on their site has a checkbox that can't be checked – and this is a consent checkbox, so the form is effectively unusable.

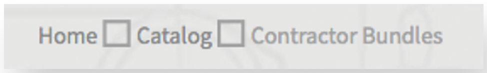
Take inventory: We go to the page to find the contact form and see it is in an iframe supplied by a vendor of this customer. At first glance, this doesn't seem to be a problem that we can fix.

Since we want to avoid reporting an issue with their iframe if there really isn't one, we are going to investigate a little more before notifying the vendor.

We confirm that the issue is not something specific to our integration with the iframe. We also look at other similar pages on the site and notice the same problem with the iframe on those pages. We look at one of our forms that is similar, and it functions correctly. Since we haven't found any places where our code doesn't look right, and because we have no direct control over the iframe, we notify the vendor of the issue and tell them this is likely a significant problem affecting all their websites.

Boxes instead of brackets

Customer tells us the ">" symbol between their breadcrumb entries (just under the header) is showing a square box "[ ]". What was strange is the ">" symbol did render as expected on some pages but not others.



Take inventory: Experience will tell you these square boxes are likely a result of a missing font character (think emojis) or a missing font. In clicking around on the website, we found the pages on which the symbol worked contained "www.[website name].com" whereas the pages on which the symbol was broken were "[website name].com". We can open the Network tab (or the Console, for that matter) in Chrome Developer Tools (see Chapter 9) and see a font file showed a 404 error.

Attempt a fix: Since we know this font file loads on the "www" domain but not the non-"www" domain, the fix is now apparent. We missed an item on our launch checklist to ensure all requests are redirected to the primary domain name (this could be either "www" or non-"www", but it *must be* one or the other). The problem is solved.

Thus far, you would rightly consider these to be easy fixes.

FOUT in unsightly places

A customer requested we look at the loading sequence of their website. The site had a pretty bad case of Flash of Unstyled Text (FOUT). Text would appear in a default system font – and then it would change to the nicely-formatted font. The delay was apparent and ugly.

Take inventory: There were requests that were blocking the loading of the applicable stylesheets (and fonts). The fonts were downloaded from Google and using `font-display: swap`.

Attempt a fix: Network Inspector (in Developer Tools, in Chrome or Firefox) is a must-have debugging tool you should know how to use. This showed us what we could change to improve font loading times and render strategy preferred by a client.

These are one of the most difficult tiers for a junior developer to troubleshoot because there seems to be no apparent way to slow down these actions so you can investigate and get a picture. In fact, there actually is, and we will look at this in detail in Chapter 9.

An event is not being triggered as expected

The idea with most of the problems above is, "Code is run, but there is a problem with it." We break into a new category where either the code is *not* run or functionality is missing.

As mentioned above, while the number of these issues is often fewer, the amount of time to track them is proportionately higher. If the project is of any size, these are daunting issues. Cracking these cases will contribute toward your "senior developer" designation.

Error pages that shouldn't be error pages

One of our customers reported that, after they created a new product, the product's page showed a 404 Page Not Found error. This sure seems to fit well into the category where "an event is not being triggered as expected."

Take inventory (round 1): I start with what I know. Their site uses a caching mechanism called Varnish. I can tell this page was frozen in the Varnish cache with that 404 Page Not Found error because the response times are miniscule. Interestingly enough, I can clear the Varnish cache and the product renders as expected. This means the page is not cleared from the cache.

Armed with information, I could tell the client to go flush Varnish every time they release a new product … or wait 24 hours and the product will appear on its own, since pages only last so long in Varnish. I didn't do that.

I looked in two places: (1) Either the request to clear the page is not being sent or (2) Varnish is not taking action on the request. I verified the cache clear request was automatically being sent when the product was saved. I also verified Varnish would clear pages on request.… So where should I look next?

Somewhere in here, I jumped to the **do it again** step, flushed my brain, and got back to work.

Take inventory (round 2): I involved the hosting company to see if they had ideas. Yes, they did (it is good to get second opinions!). We added the page's identifier (cache tag) into the response headers. Would you believe the page identifier for the 404 Page Not Found error had no references to the product? When the product saved, the cache clear request was triggered for this product's page identifier, but nothing was found. We could easily chalk this up to a core bug.

Attempt a fix: I created a plugin to add the product identifier to the 404 Page Not Found (when this product was intended to be loaded), and the problem was solved.

Back in stock... but maybe not?

This same customer also relies heavily on "back in stock" notifications being sent to customers. We rebuilt part of this functionality to make it asynchronous – emails would be queued and sent outside of the current web request. This makes the "receive inventory" operation significantly faster.

It tested well. We received out-of-stock notifications.

We deployed the code to production thinking this was another job well done.

Until ... we found only *some* out-of-stock notifications were being sent. GRRRR. If an email was not sent, that's something we can troubleshoot, but *only some*?!

Take inventory: In situations where there are many components, it is helpful to break them down into small pieces. Instead of giving steps to sending a "back in stock" email, I'm forming them into questions, which helps me think of potential problems:

- Is the code recognizing this product is out of stock?
- Is the code properly calling the asynchronous action?
- Is the message queue calling the code to send the email?
- Is there a problem with the email?

While these specific questions are likely irrelevant to your situation, I hope you see how important it is to identify *every* piece of the puzzle and test each piece individually. By identifying them, we can ensure each step is successful and isolate it. So in my situation where the emails were not sending, I used the

Learn to categorize the problem as you approach it. Great developers remember fewer specific solutions than general ideas.

above questions to help me. The problem came down to code for a specific situation that was triggered but not handled properly.

Attempt a fix: I wish I could say my first fix worked, but it didn't. A challenging aspect was the code to trigger emails was very convoluted, and I couldn't figure out why. Instead of trying and trying to make it work, I ended up rebuilding the mechanism for formulating the email. I also added solid error handling. The final solution is a significant improvement.

When troubleshooting problems where there are a significant number of steps to replicate, consider how you can simplify the process. For example, in the case above, I wrote a CLI command to trigger an email through the message queueing system OR by directly calling the email send code. This simplification helped me reduce the cognitive overload when troubleshooting, and it allowed me to iterate much faster.

Performance problem

We are now into the mother of pearls category. I find performance problems the most difficult to troubleshoot ... unless I have the proper tools with which I can identify the problem.

The culprit of this category is usually *too much code* is run. Necessary hooks often trigger other unnecessary hooks. While you should strongly pursue building highly performant applications, we would all agree that some things will escape even the very best developers. Or, more likely, we will be enhancing the performance of others' code.

What makes this especially difficult is that the cause is often buried. One process triggers another, etc. I hope you use a solid local environment because you will need it.

Saving a product is slower than a turtle race?

A customer notified us that when they disassociate several categories from a product, the product save operation was slow.

Take inventory: Antonio worked on this problem. This is a challenging one because there are so many processes that are triggered – where do you start? While he began with his debugger, another tool to use is a profiler (Blackfire, Xdebug profiler, or even New Relic).

The end result is that he located where category relationships were changed and stepped through this code, line-by-line, in his local environment. He identified most of the processing power happened when processing the associated URLs for this product + categories. He observed that occurred in an indexing process.

Attempt a fix: In Magento, there are a couple of indexing process modes, one of which is asynchronous. We should not be modifying core URL processing. Our hypothesis is that by switching this indexer to asynchronous mode, the intense processing will be offloaded to a backend process and will no longer affect administrator efficiency.

What is it about saving products?

A customer asked us to look at why saving products was so slow. Sometimes, they even got a timeout error. There didn't seem to be a correlation between the changes that were made and the length of time to save.

Take inventory: This is one of those problems to which the solution could be almost anything. There are likely hundreds of thousands of lines of code that are run when saving a product. Because there was an occasional timeout error, I started looking in the log files. Here is an entry I saw:

```
[0x00007f3532a1f220] process() /srv/releases/build-783125522/app/code/
MyCompany/Xnotif/Model/Observer/SendEmailNotifications.php:39
[0x00007f3532a1f290] _processStock() /srv/releases/build-783125522/
vendor/magento/module-product-alert/Model/Observer.php:428
[0x00007f3532a1f300] sendNotifications() /srv/releases/build-783125522/
vendor/third-party/xnotif/Model/Observer.php:638
[0x00007f3532a1f460] loadProduct() /srv/releases/build-783125522/
vendor/third-party/xnotif/Model/Observer.php:197
```

This may or may not be the problem, but it's a place to start. Luckily, in reviewing the code, I found that every single out-of-stock notification was iterated (there are tens of thousands of these) and every product for each notification was loaded (a very CPU-intensive operation). This all happened *every single time the product was saved!* The third-party module got the job done, but it was poorly built.

Attempt a fix: You may recall that, in a previous section, I said we made this functionality asynchronous. We did, AND we also filtered this collection so that it was only looping through records associated with this specific product that was saved instead of ALL products.

But what if we don't have log files to point us to the problem?

- Capture the state of the website. See Chapter 8 for suggestions on downloading a copy of the production database.
- Utilize a profiler so you can see every method that is called and examine the memory used. The hardest thing to catch is a loop that occurs through several frames. If you look for it, you might just find it.

    - If you’re not able to observe the issue while you’re profiling the product save, you can use a performance monitoring tool like New Relic or Tideways. Those tools can capture high-level profiles of slow-loading pages, which may help your troubleshooting efforts.
- Consider if there are dependencies that can have an effect. For example, are you aware of your website calling any external web services? The easiest way to write these calls in PHP is synchronous, so the application must wait for a response. If a timeout is not properly selected, there is a chance the external web service is experiencing an outage, which will have a ripple effect. I used to see this regularly with live shipment rate calculations: the API would go down, and so did the customer’s checkout system.
- Do your best to identify every component that is triggered. If you can, separate the pieces out and test individually. If this is not possible, keep mental track of which piece you are troubleshooting.
    - The expensive processes aren’t always immediately visible. For example, you might have an on-the-fly JavaScript bundling system that runs on some requests but not others.
- Replicate this in your local environment.

Security-related

You are bound to run into a data breach – or at least an attempted one. If you don’t believe me, go watch the access logs for your website and you will likely see thousands of entries for bots looking for vulnerabilities (usually SQL injection).

Much of debugging is forensics. You have some information to work with, and you must figure out the solution. Security-related problems fall into this same camp but tend to rely even more on logs than anything else. Reports of data breaches are often vague and can be found by browsing the database and finding something that looks “off.”

My biggest pieces of advice are:

- Write secure code. Know the OWASP Top 10. Many security vulnerabilities are simple mistakes that are easily avoided. And, for Pete’s sake, please do not use easy passwords (`asdf1234` is not a good password; and `asf8923uyajk!!!` used on all of your logins is also not good).

- Consider the “blast zone.” If a malicious third-party were to gain access to *something*, what damage could they do? For example, do you have all of your hosting management under the same account? If “yes” and your credentials are compromised, the attacker could then gain access to a devastating number of websites. Does your staging website have direct access to production?

- Have a plan for when the inevitable happens. This should include any companies who will be part of your emergency team. Verify your backup files. Ensure that only minimal dependencies are enabled on the server. You know the drill.

- Save any and all evidence. Don’t delete anything. If malicious code was injected, make a backup of the old code and deploy the website again (to reset all changes).

- Get a professional to help. It’s okay. Things happen. This might be a big enough deal that you need someone who has resolved these situations in the past.

- Assume the most obvious attack is a red herring. A few years ago, I received an email from one of our clients saying he was unable to log in. The easy/default answer would be to say “reset your password.” But I looked into it and found his password was recently reset. After reviewing access logs, I recommended that they reset their passwords. Yet, the question was burning as to how someone got access. We engaged some security professionals who discovered there was a module that had a security vulnerability.

Chapter 8

DEBUGGING STRATEGIES

So I couldn't replicate the cart freeze issue your customer experienced the other day. But it seems to be fine now!
I guess it's like a tree falling in the forest – if no one's there to see it, did it really happen? Haha!
Yes, but the tree fell on my business.
I'll find the issue ASAP.

This chapter will focus on practical methods for solving problems. You have graduated from the first level in detective school and should be fired up, ready to begin troubleshooting. At this point, though, you likely don't have the tools to sniff out the problems. Sherlock Holmes developed his expertise over many years.

Replication in your local environment

"If a tree falls in the forest and nobody heard it fall, did it really happen?"

I do not mean to get into a philosophical debate, so please come to your own conclusions about this subject (Wikipedia has a great article on the subject). But for the sake of this book and your legacy as a developer, I hope our answer to this question is an emphatic "YES." Things we are not aware of happen all the time in a production environment.

For us to effectively fix a bug, we *must* replicate the problem. If we can't replicate it, our tendency is to throw it back to the merchant and say, "I can't duplicate this problem so there must not be one."

Hopefully after reading the previous chapters you can agree we will use our brains and will do smart research to gather as much information as possible before going back to our client. This is a part of the first characteristic of a great developer (from Chapter 4).

Why local replication?

Your local environment has the tools to effectively debug problems (see the next chapter). You make a change, and it immediately appears. You know your changes will cause no harm to any other environment. You are comfortable in this environment as it is one environment, not the many environments on which you manage production websites.

If you're not using a local development environment and are instead using a development or staging server as you write code, I would highly recommend spending the time to configure a local environment. Your efficiency will likely increase by 2x at least.
See swiftotter.com/artofdebugging for local configuration recommendations.

Is it impossible to solve problems in a production environment? No. But it's not ideal. You may have to do this as a last resort. Over time, you will develop intuition about when it is appropriate to move your troubleshooting to production. One of these examples was the situation I mentioned in Chapter 2 where I was unable to quickly replicate the problem locally. A big marker on that particular issue was this problem didn't appear on staging before going to production. But, remember, computers are not people, environments can be cloned, and your staging environment should be a near clone of production. Problems that arise in production almost always come from differences between staging and production.

Always keep your local environment up to date

I'd recommend creating a shell script (committed to your project) that allows you to easily sync the production or staging database down to your local environment.

It should take you a few minutes to refresh your local database. For many systems, you simply SSH into the production environment and run something such as this (on Mac):

```
ssh Production-Server
mysqldump -u[user] -p [db_name] | gzip > [date, like 2020-12-19.sql.gz]
exit; # get the SSH out of here

scp Production-Server:2020-12-19.sql.gz
gunzip 2020-12-19.sql.gz
mysql local_db < 2020-12-19.sql
```

This is super easy to run, and you can mostly commit it to muscle memory (except for the passwords). Gzipping the result reduces the amount of time to download and it only takes a few extra keystrokes.

The above does nothing to sanitize or delete customer information in keeping with GDPR compliance. You must follow your company's guidelines and local regulations for ensuring proper procedures are followed. The less customer information you have in your possession, the better. If it is difficult to keep fresh databases on hand, consider using something like Driver (github.com/SwiftOtter/Driver) to securely and safely automate the process.
You can also go to swiftotter.com/artofdebugging for more platform-specific information.

Please, always, always make sure this downloaded database has been sanitized from all keys that are only supposed to work on production. Otherwise, changes you make in your local environment *will affect production.*

Examples would be search API keys for services such as Algolia and payment API keys such as Stripe or Braintree. Search is especially painful as you might override indexed entries on production, pointing customers to your local development environment's URL—not good.

Differences between your local environment and production

I hope your efforts immediately uncover the path to replication in your local environment ... but, in all reality, it won't always be that way. It isn't for me. Even if your database is 100% up to date, you might still be missing something. Let's look at the key elements of running an application and see how they can introduce differences:

#1 Code

Your local environment might have different files or versions from those in production.

Do you have a build pipeline that might introduce inconsistencies? Are you sure there are no differences in branches (say, between the `release` and `master` branches)? If you are using an ecommerce platform that requires compilation, have you run those exact same steps locally? Or have you straight-up downloaded the code from production to your local machine?

I highly recommend a build/deployment process where the code is considered immutable. This code is first delivered to the staging environment. When you are ready, it is deployed to production. This 100% eliminates any *code file* differences between staging and production (you could still have images that are different).

Some hosting systems take this to the highest degree where the filesystem is read-only and the deploy process literally builds new containers and deploys those. Even if this is not your environment, I strongly believe it is the best practice where you can automatically drop a new set of code into production, and it will run. If nothing else, this is good for security as breaches that affect the filesystem can be immediately reversed (first, you must make a backup of the website for forensic evidence).

#2 Database

This is the easiest to resolve as you should be able to quickly download a new database. If there is an error on production and you are unable to immediately replicate it in your local environment, your next step is to download a new version of the database.

Even if you have the most up-to-date database, there still might be something you are missing: the context. For example, you were given the information that a specific error appears when logged in as a customer. What's missing? The customer's email address. Why is this important? What if they are configured as a B2B user with a specific company account and your user account is configured as a B2C user?

#3 Operating system

If you run into a difference on the operating system, you've got big problems. These are nasty to resolve and take time. It's not impossible to resolve, but it requires a methodical approach to isolate what is going on. I find these problems rare in small-to-medium applications and only become more of a problem in enterprise-level systems.

Here's a difference that will help you keep your hair on your head, where it belongs (instead of stress-induced hair loss ... come on, it's real!): case sensitivity. Most (or all?) Unix-based web servers are case sensitive. Mac and Windows machines, by default, are not case-sensitive. Back in my early days, I can't tell you how many times I created a file, deployed to staging, and it broke. The file was there. But due to a difference in the filename's case, the class would not load.

The biggest culprits are three-word filenames, such as "TriggerArtProof.php." It's easy to miss capitalizing the "A."

If you're doing local development directly on a Mac using a tool such as Valet+ (and not a virtual environment like Docker or Vagrant), you should make sure to set up a case-sensitive partition for your development environment.

The easiest antidote is to use a development environment as close to your most popular production environments as possible. For example, does your production stack use Nginx or Apache? Mirror that. Does your production stack use ElasticSearch, RabbitMQ, Varnish, etc.? Ensure those are in your local environment and they are configured (Varnish is difficult for some environments, though). Brew makes installing these in Mac environments quite easy.

If you are dealing with something unexplainable, review the versions of related packages. I experienced odd behavior in my local environment and found that I was using MySQL 8; but MySQL 5.7 was on the server.

Another perspective on these third-party dependencies is that most of them derive their data from the codebase and database. As long as the version numbers match (within reason), and you reindex/refresh/clear the cache – whatever – you should be able to replicate.

Replicating CLI commands

Good developers know that some processes take significant CPU cycles to run. They do their best to offload these into the background so as not to slow down the web response thread. In other cases, you can parallelize multiple tasks at once to reduce the time to respond back to a request.

Either of these situations can make finding *what to replicate* very difficult. You don't know *what code* is triggered. Or maybe you know what code is triggered but can't find out *how it is executed*.

Logging is your friend here. If you are using PHP, write all values from the `$argv` variable or the `$_ENV` superglobal to a log file. You can also search your codebase for the primary methods to call system commands (in PHP, this would be `system` or `shell_exec`).

Debugging APIs

Chances are high your application relies on a third-party web service in one way or another. If you accept credit-card payments, you rely on a payment gateway. If this gateway goes down or takes a long time to respond or responds with an incorrect response, how do you troubleshoot this?

When building an integration with a web service, always take into account every exception you can imagine—and then some. For example, what happens if the web service becomes unavailable? What happens if it takes 30 seconds to get a response? What if an error is returned ... when you are expecting JSON?

You should discuss these considerations with your merchant. When you properly consider them, you will reduce the number of problems you will have to debug in the future.

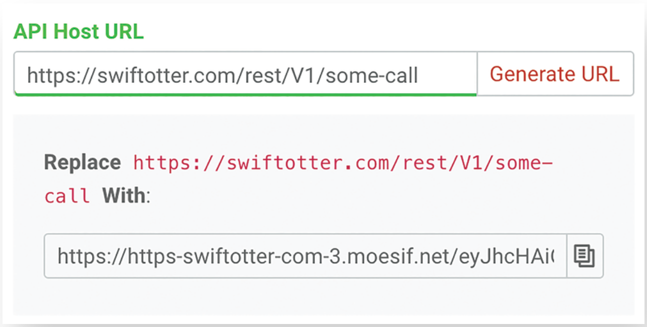
APIs are notoriously difficult to debug because the information that is transmitted is often specific only to that environment. For example, we helped a company integrate a work-automation system that moved orders from Magento to Netsuite. This work-automation tool made calls to Magento, and I isolated the problem down to these calls. But how was I to see what was in these API requests? The first, and ideal, solution was to use this tool's integrated logging experience – except that didn't exist in any meaningful way.

Option #1: Add an internal logger for API requests

You can add a component to log incoming requests. This can be fairly easy to add – if you aren't in a hurry and can deploy code. At the most rudimentary level, you would take all HTTP headers and the POST body and write that to a file. Trigger the request, and you will get a picture of what is happening.

Option #2: Use a third-party tool

My mind was blown when I came across Moesif. Moesif is a "codeless proxy." You provide an endpoint URL, and it gives you a new URL:



https://swiftotter.com/rest/V1/some-call

Replace `https://swiftotter.com/rest/V1/some-call` **With**:

https://https-swiftotter-com-3.moesif.net/eyJhcHAi(

In this case, I would be configuring an external service that calls swiftotter.com. Instead of entering the “swiftotter.com” URL, I add the “moesif.net” URL. This external service will call the “moesif.net,” the request is logged in its entirety, “moesif.net” then transmits the request to “swiftotter.com.” The response travels back through “moesif.net.”

You can even repeat these API requests and modify data. You can also reverse the order and send requests from your application to the third-party web service *through* Moesif.

Please be careful with sensitive customer data. You will be logging data and this could include personally identifiable information. There can even be a few (and very bad) cases where customers’ credit card information could be passed through a tool like this.

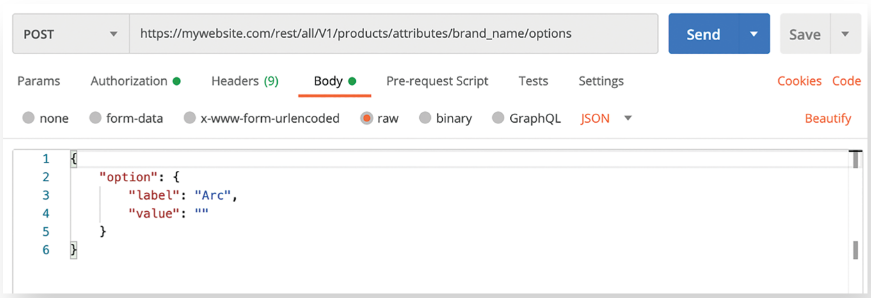
Postman

Remember, we are always on the lookout for shortcuts or simplifications of the debugging process.

Picture this with me: You are troubleshooting an operation that saves product information into a third-party search index (e.g., Algolia). The problem is the authentication is broken. You are sure the correct credentials are stored, but this third-party system is not accepting them. You could keep tweaking your code and refreshing the page to kick off the save operation ... again.

Or you can use a tool where you can build a case study for what doesn't work. Package it up and send this information to the third-party's support team: Postman is your friend. As of the time of this writing, Postman remains a free API development app that works on Mac, Windows, or Linux.

Postman allows you to craft requests. You can configure the POST body with JSON. You can set headers. You can provide inputs for Oauth access. Then, you can save these requests so you never lose the command. You can export as curl to associate with a work ticket – forever.



```
{
    "option": {
        "label": "Arc",
        "value": ""
    }
}
```

Get the stack trace

A stack trace is the list of method names that arrive at the place where an exception is thrown. Here is an example:

```
NOTICE: ARRAY TO STRING CONVERSION IN /SRV/RELEASES/BUILD-796713766/
VENDOR/LAMINAS/LAMINAS-CRYPT/SRC/UTILS.PHP ON LINE 31

#1 Laminas/Crypt/Utils::compareStrings() called at [vendor/magento/
framework/Encryption/Helper/Security.php:28]
#2 Magento/Framework/Encryption/Helper/Security::compareStrings()
called at [vendor/magento/framework/Data/Form/FormKey/Validator.php:39]
#3 Magento/Framework/Data/Form/FormKey/Validator->validate() called at
[vendor/magento/framework/App/Request/CsrfValidator.php:76]
#4 Magento/Framework/App/Request/CsrfValidator->validateRequest()
called at [vendor/magento/framework/App/Request/CsrfValidator.php:130]
#5 Magento/Framework/App/Request/CsrfValidator->validate() called at
[vendor/magento/framework/App/Request/CompositeValidator.php:40]
#6 Magento/Framework/App/Request/CompositeValidator->validate() called
at [vendor/magento/framework/App/FrontController.php:160]
#7 Magento/Framework/App/FrontController->processRequest() called at
[vendor/magento/framework/App/FrontController.php:118]
#8 Magento/Framework/App/FrontController->dispatch() called at [vendor/
magento/framework/Interception/Interceptor.php:58]
#9 Magento/Framework/App/FrontController/Interceptor->___callParent()
called at [vendor/magento/framework/Interception/Interceptor.php:138]
#10 Magento/Framework/App/FrontController/Interceptor->Magento/
Framework/Interception/{closure}() called at [vendor/magento/module-
store/App/FrontController/Plugin/RequestPreprocessor.php:99]
#11 Magento/Store/App/FrontController/Plugin/RequestPreprocessor-
>aroundDispatch() called at [vendor/magento/framework/Interception/
Interceptor.php:135]
#12 Magento/Framework/App/FrontController/Interceptor->Magento/
Framework/Interception/{closure}() called at [vendor/magento/module-
page-cache/Model/App/FrontController/BuiltinPlugin.php:71]
#13 Magento/PageCache/Model/App/FrontController/BuiltinPlugin-
>aroundDispatch() called at [vendor/magento/framework/Interception/
Interceptor.php:135]
```

```
#14 Magento/Framework/App/FrontController/Interceptor->Magento/
Framework/Interception/{closure}() called at [vendor/magento/framework/
Interception/Interceptor.php:153]
#15 Magento/Framework/App/FrontController/Interceptor->___callPlugins()
called at [generated/code/Magento/Framework/App/FrontController/
Interceptor.php:26]
#16 Magento/Framework/App/FrontController/Interceptor->dispatch()
called at [vendor/magento/framework/App/Http.php:116]
#17 Magento/Framework/App/Http->launch() called at [vendor/magento/
framework/App/Bootstrap.php:263]
#18 Magento/Framework/App/Bootstrap->run() called at
pub/index.php:53
```

Reading stack traces is more boring than reading the instructions on the back of a paint can (which is surpassed only by watching the paint dry). Even worse, they are filled with genealogy-esque "words" that can make any junior developer quite anxious if they are forced to look at it (at least it did for me).

To the great developer, call stacks are the key to solving problems; they are the weapon with which we slay the gremlins that live in our application. I find these to be the single-most-useful source of information for solving a problem. Once I get my hands on a stack trace, I am now in my comfort zone. I feel confident we are fairly close to a fix.

When is a stack trace helpful?

Stack traces are useful *when you don't know how this happened.* You must have a solid idea of *where this happened*, often through an exception that is thrown. Stack traces help you work back and reverse-engineer the path of methods that made this problem happen.

For example, let's examine the stack trace presented at the beginning of this section.

```
Notice: Array to string conversion in /srv/releases/build-796713766/
vendor/laminas/laminas-crypt/src/Utils.php on line 31
```

This error is one small step better than being useless. By examining the call stack, we see the error is ultimately triggered by incorrect information coming from:

```
Magento/Framework/App/Request/CsrfValidator->validateRequest
```

The call stack gives us critical context.

How to use the stack trace

I use stack traces to set breakpoints in my local debugging environment (see Chapter 9 for information on using a debugger).

If you are not able to replicate this exact problem in your local environment, consider logging these stack traces along with information to build the context.

How to get the stack trace

Most every language has a mechanism for retrieving the trace. Use it and log it.

PHP:

```
debug_backtrace(DEBUG_BACKTRACE_IGNORE_ARGS, 100);

// or with an approach that utilizes the PSR-3 LoggerInterface
try {
   throw new \Exception();
} catch (\Exception $ex) {
    $this->logger->critical('Exception thrown', [
        'trace' => $ex->getTraceAsString(),
        'product' => $product->getSku()
        // ...
    ]);
}
```

But what about the frontend? Yes, JavaScript has stack traces, too:

```
console.trace()
```

We use Sentry.io on all of our frontend applications. If you think it's hard to locate useful information from backend logs, it's doubly-difficult to get helpful logs from the frontend. On the backend, we have an easy central place for logging – the files. You SSH in and can review them. But browsing the frontend is like searching a silo. Errors happen, the application is unusable, but errors disappear into oblivion as the user navigates to the next page.

These tools are a lifesaver. Errors are logged up to Sentry, along with the stack trace. You can even configure alerts to stay on top of any potential problems.

Ensure we are on the same page as the merchant

Have you ever had a seemingly intelligent conversation with someone only to find out you are talking about something completely different? My two-year-old is especially adept at this. It's laughable.

We live in a world where communication is increasingly virtual. Messages are entered via a keyboard, or worse, a smartphone. There are still many people who don't type very fast. Some of us (I raise my hand) cannot *quickly* put our thoughts into words in a way that is easily understood.

Let's say we have a problem report like this:

> When some orders are submitted, they don't have the correct IP address specified.
>
> OR
>
> When I save a new product, I set the correct inventory level, but it still is recorded as out of stock.

In these instances, our first step to *Take inventory* is to identify how this happens. We would probably submit an order or save the new product. But let's say this works as expected. What then?

After I have exhausted my list of ideas, I go back to the merchant and ask exactly *how* they received this error. For example, they could be saving the order through the admin panel, or synchronizing the order from a third-party marketplace such as Walmart. They could be loading the product through an Import tool.

We are human, and we will not think of everything. Don't hesitate to ask for clarification when necessary.

Consider calling a meeting with your client instead of asking them to again type up what they didn't want to type up in the first place – the steps to replicate the issue.

Side-by-side comparison with a system that works

Have you had the opportunity to debug those strange issues when an order is placed? I swear this is one of the most challenging areas to debug. There is a ton of complexity that happens here. To make matters worse, there are often third-party plugins with their fingers in the mix.

The other day I was talking with Erik Hansen, and he mentioned this as a strategy he often uses. Frankly, I haven't used it as often as I could, but I'll share the story of one time I did and it was very helpful.

Example:

```
Property "DisableTmpl" does not have accessor method "getDisableTmpl"
in class "Magento\Quote\Api\Data\PaymentInterface"
```

The error was thrown on the server when placing an order. The source of this error is the frontend. Many people are quite unfamiliar with how orders are placed, so the idea of debugging this can be daunting.

We will cover it in-depth in our next chapter. In the meantime, here is an overview of how I would approach this problem:

1. Open Chrome Developer tools, go to the Network tab.
2. Locate the request to place the order.
3. Navigate to the Initiator tab and find the place in the call stack that looks like a possible trigger (often there are a number of frames that build and send the request to the server).
4. Set a breakpoint at this location, and others as necessary.

You might collect data – but identifying the data that is causing the problem is likely a bigger problem.

This is where having a vanilla environment configured is helpful. Set a breakpoint in the same location as in your local environment and trigger the identical sequence of events. Follow the breakpoints through, simultaneously. You will easily spot the differences.

This only works in some cases as third-party modules can make significant differences in the logical flow. Even then, you are not out of options.

Do you have another local environment that has this specific module? I find agencies generally stick to the same module for a given use case.

You can also quickly zero in on third-party modules, or your customizations, that modify core functionality. This also helps you to isolate potential sources for this problem.

Identify contributing factors

We recognize that many aspects can contribute to “this problem” happening. When we figure out the correct steps, we have then arrived at a replication.

Unfortunately, we don’t live in an ideal world, and we don’t always have a beautiful, clean list of steps. The replication process often fails because we are unable to find “that trigger” for “this problem.” Because we are unable to replicate the problem, the ill-advised tendency is to go back to the merchant saying “this is not a problem because we cannot replicate.” Remember, this only creates frustration from the merchant’s perspective.

We must put our detective hat back on and identify the differences between the reported situation and how we triggered the problem. While the list could be infinite, write down the most probable options. Order the list from the most probable to the least probable.

Examples of contributing factors

Capture the flag. In Chapter 4, I discussed a problem where an exception was thrown. Turns out, this code was triggered if the customer had a specific Tax Exempt flag set in their account. Quality Assurance probably didn’t have this flag set, so their test cases passed with flying colors.

Too many cookies give a stomach ache. We have occasionally run into problems where Magento throws an error about “too many cookies.” The site works flawlessly for us but not for the visiting customer. What could be the difference?

The web is stateless. When we first visit a website, there should be absolutely no difference ... except for IP address (yes, and this can be a problem). As we begin using a website, the information is stored in cookies or `localStorage`. Differences between browsing sessions quickly begin to arise.

There are many tools such as the Cookie-Editor or EditThisCookie Chrome plugins that package up cookies so your visitor can send you a more complete picture of what they are experiencing.

Frankly, if you run into enough problems here, it's not that hard to build a troubleshooting page that dumps this information so the visitor can easily copy and send to you. You would want this page to be protected from the general public so as to avoid the possibility of Session Hijacking attacks (where a malicious third-party steals your cookies and "logs in" as you).

Comparing the columns. While obtaining cookies is an excellent means of troubleshooting, it might not always be possible to get the list or to use them (for example, if you have session/IP restrictions in place so that a session is only valid for a given IP address).

In these cases, I manually compare two entities: the one that works against the one that is broken. Column by column or table by table, you should be able to identify any differences.

Not even Cloudflare is perfect (but it's really close to perfect). I recently surveyed almost 130 developers about their experiences. Here is one of the responses I would like to share.

He worked on a problem where customers were sometimes not able to login. This word "sometimes" is one of the worst things you can hear as a developer. There were no logs anywhere for this. It turns out that the Speedy feature was enabled in Cloudflare. This created additional cookies and sometimes prevented customers from logging in. He reported this to Cloudflare, and they applied a patch. He then added a JavaScript snippet on the website to clear the affected cookies.

Dealing with random problems

If there was one thing that is the most hated aspect of debugging, it would be fixing "random" problems. What's the solution to something that only happens occasionally? For the sake of clarity, I consider a random problem as something that you can replicate *sometimes.*

Jonathan Lorenzi put it very eloquently when he said that he doesn't see random problems as random. Computers accept instructions and input from human beings (or monkeys for that matter, or guinea pigs, or ...).

I have found it helpful to assess the contributing factors for a random problem. What could cause this error? In reality, errors are handy in that you have a specific place where you can attach logging and obtain a stack trace.

But what if it is data that is randomly missing? Write entries to the log file (see Chapter 9).

Examples:

Race conditions are *not good*. If you occasionally get a JavaScript error when loading a page (jQuery anyone?), you are likely looking at a race condition (when one script is loaded and executed before another script, on which it depends). For example, if you initialize a particular JavaScript class in your HTML, but do not wait for the document's `DOMContentLoaded` event, you run the risk of throwing an error where this class is not found. This will often be random because your JavaScript class should be cached after the initial page load, and it will not be recognized until the cache is cleared.

The best solution is to use a JavaScript dependency-management system.

Yes, you learn this through experience or by reading *The Art of Ecommerce Debugging*.

Sessions can cause problems, too. We took over a "rescue" project. The previous agency was not able to deliver a website that functioned as the customer needed. One of the more onerous problems that lingered was where database records were randomly dropped. The customer said, "If we have no database stability, we have no website." I agree 100%.

Of course, I couldn't replicate it. They couldn't either. They would be working and notice that some of their previous work was missing. That doesn't make the problem any less serious.

My first approach was to review the saving process. How did it work? I found something quite interesting. Identifiers that were stored in the session were used in the saving process. *What would happen if I opened Page A in one tab, then Page B in another tab, then went back and saved Page A?* Wouldn't Page A be saved with Page B's ID? Wouldn't this cause the problem?

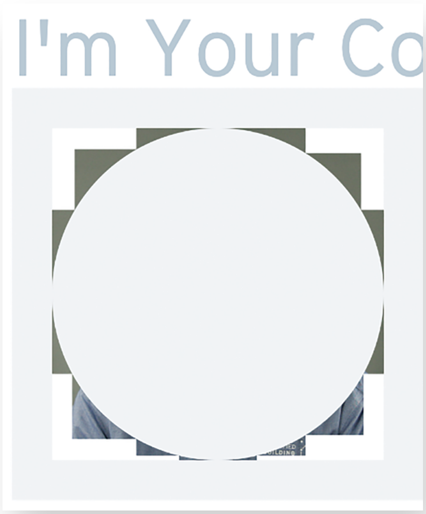
This was the problem, and the solution was quite simple.

These are two examples where we were able to fix the problem immediately. In most cases, we aren't so lucky. Your go-to tool is logging. Use it liberally.

When did this start happening?

I find that very few problems "happen overnight" (those that do are usually related to a consumable resource such as hard-drive utilization). I'd estimate that 80% are directly attributable to code changes. If your company uses version control (and they should be!), you will have a specific commit associated with every problem.

A great example of this is with Composer (a package management system for PHP). DomPDF is a PHP package for generating PDFs that we include through Composer. Before a code delivery, our QA analyst noticed that images looked like this:



The image is reversed in that only what's inside the circle should be visible—someone's smiling face.

How should we figure this out? I am no expert on image processing.

We start by understanding *what changed.*

Take inventory: I review commit logs and see we updated Composer packages and DomPDF was upgraded. DomPDF is responsible for building the PDFs. The image hasn't changed, so the fact that DomPDF was upgraded makes it a likely suspect.

Attempt a fix: The immediate solution is to downgrade DomPDF by locking it in at the specific/previous version. Images are now rendering as expected.

Do it again: Now that this problem is resolved in time for our code delivery, we must step back and repeat the cycle to obtain a long-term solution. We must identify the differences between these two versions and determine what was causing the problem. We would then either:

- Revert that specific change. We would look into the pull request in the website's repository to understand why this change was made.
- Research to figure out why this change is incompatible with the server environment. It could be a difference in software dependencies – perhaps DomPDF is expecting a library to be installed of X version and Y version is actually installed.

Identify how this could happen through a review of the code

There are some cases where we are genuinely unable to figure out what is causing a specific problem. I find walking back through the code, line-by-line, to be helpful. This works best when you have a good understanding of the structure of the code and a general idea of where this problem would happen.

To make my code review more effective, I try to replicate the error as closely as I can. I step the debugger into this area and observe the variables in the debugger watch panel. I then ask the question, “What variable value would need to change to facilitate this malfunction?” This is somewhat of a last-resort method, but it has worked for me in numerous situations.

Process of elimination

Sometimes you will encounter issues that don’t result in any specific error messages, and you don’t know how you’re going to get to the bottom of the issue. A helpful technique is something I call “process of elimination.” You can disable all third-party and custom extensions and see if the issue goes away. If it does, selectively re-enable groups of extensions until the issue reoccurs. Once it does, disable extensions until you find the culprit. You can then take a similar approach by commenting out or disabling code within the problematic extension until you find the source of the issue.

Don’t rule out core bugs

But also don’t first assume you are dealing with a core bug.

I remember back to my elementary school days and, specifically, my math studies. I would tell the teacher (who also happened to be my mom) that my answer to the math problem was correct … and the answer key was wrong. While software is nowhere near as perfect as math answer keys, I still think this proves my point quite well.

The first step to identifying a core bug is to go to the platform’s GitHub page. Search the issues or pull requests for the error or a particular phrase. A good number of the bugs you may encounter will already have been documented, and many will have solutions.

To see if it is a bug, spin up a vanilla version of this software and test the process side-by-side to see where the differences appear.

I ran into an issue where an error was triggered when running a command via the CLI. This did not happen in the admin panel when running the same action (our minds should immediately be jumping to the environmental differences between the CLI and the admin). Yes, it turns out that my assumption was correct: A necessary piece of information was specified for only the admin and not globally. I changed the scope of this information, and the problem went away. Was that a core bug? Yes, from the fact this error occurred in the core. No, from the fact this occurred outside the logical flow as defined in the core. I still submitted a pull request because others will benefit from the fix.

If you locate a commit stored in your framework's GitHub repository, you can easily apply this as a patch. Append `.patch` to the commit's URL, and it will be formatted so you can apply it with `git apply patch`.

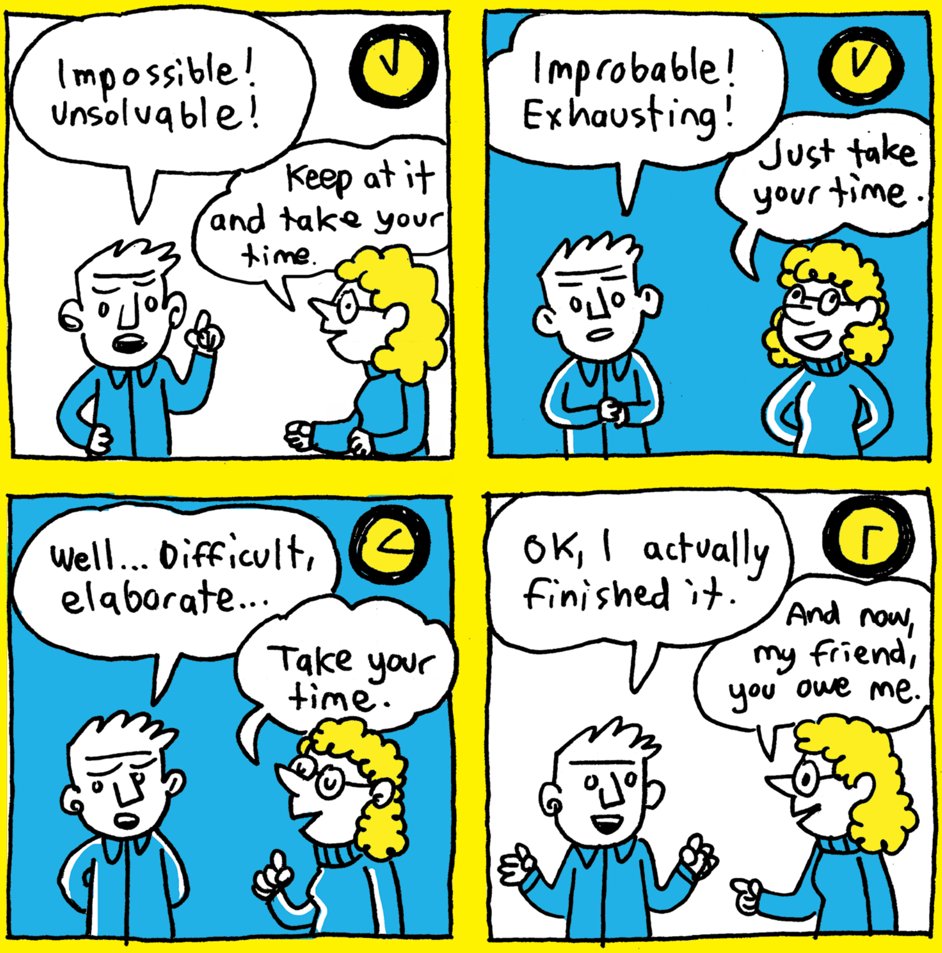
Copy-and-paste is your "frenemy"

I've seen dozens of issues caused over the years by developers incorrectly hand-typing a long string (or snippet of code) that could have been copy-and-pasted. If there's room for error, use copy-and-paste – this should prevent you from spending time debugging down the road.

On the flip side, once you do copy and paste a snippet of code, take the time to review this code. Is the context in which you use this snippet different from the previous one? If so, you could be introducing bugs.

Chapter 9

THE DEBUGGER


Impossible! Unsolvable!
Keep at it and take your time.
Improbable! Exhausting!
Just take your time.
Well... Difficult, elaborate...
Take your time.
OK, I actually finished it.
And now, my friend, you owe me.

Tools were one of the earliest human inventions. Our soft, fleshy hands can only hit a nail so hard before the pain becomes unbearable (not to mention, the significant damage to our skin). Unless you have your black belt in Karate, you probably can't modify the structure of a wooden board with your bare hands. I have neither a strong pain tolerance nor a black belt. I turn to tools.

Tools fascinate me. For example, I have a favorite hammer. It is a Stanley 16-ounce claw hammer. No other hammer compares, and I've used plenty – others just don't have the right feel. I received this hammer when I turned 13 years old (if my memory serves me correctly). Twenty years later, I still reach for this hammer. This hammer has built a house, helped with many remodels, and probably pulled just as many nails as it has driven. It has even hurt me a few times too. This hammer has made my home updates significantly easier.

Like a favorite hammer, the tools I am going to share with you will revolutionize your ability to quickly solve problems. You will learn these once, and they will stick with you forever. Yes, you will refine and learn new adaptations, but the basic ideas presented here will last a lifetime.

Consider dedicating even five minutes a day to professional development (see Chapter 4 for a conversation about this). Start by studying and practicing with the tools discussed in this chapter. Your efficiency will skyrocket.

I strongly suggest you use an IDE (integrated development environment). These have a tremendous number of tools that will boost your productivity. Debuggers, as we will discuss next, are a built-in feature.

Debugger (Xdebug)

When I started with PHP, I used `var_dump` and `die` all the time. It got the job done. Oh, and the Refresh button. I must have worn out the “Ctrl” and “R” keys on my keyboard.

Then I was introduced to Xdebug and … wow! My mind was blown the first time I saw it back in 2011. I spent hours and hours to get Xdebug running in NetBeans.

Over 90% of what I’m discussing here also applies to Chrome/Firefox Developer tools and likely to other languages. The concepts are the same. The features are the same. The only differences are the environment (and IDE versus the web browser) and the keyboard shortcuts.

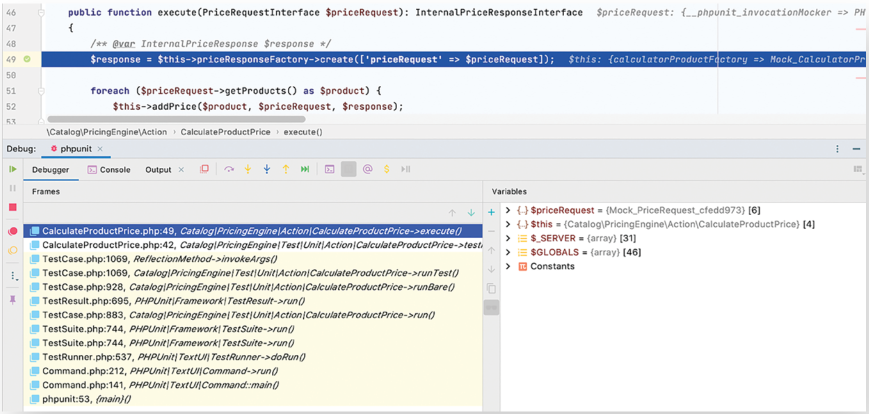
Xdebug revolutionized the way I work and the speed with which I am able to solve problems. My friend, please invest the small number of hours necessary to install Xdebug. If there were a single tool that has the potential to double your productivity, *this is it.* **I would go so far as to say that knowing how to effectively use a debugger is central to your efforts to become a great developer.**

While Xdebug is an amazing debugging tool, it will slow down your development environment substantially if you leave it enabled (even if you’re not actively using it). Come up with a way to easily toggle the Xdebug PHP extension so it’s only enabled when you’re using it.

Put simply, Xdebug allows you to step through the execution of your code, line-by-line. You can:

- Set breakpoints so the parser stops at a given location. These breakpoints can even be triggered when a statement evaluates to be `true`.
- Cancel script execution at any point. For example, you can prevent the order from being placed, which saves you from having to go through and recreate it.
- Look at, and change, the values of your variables in real-time!
- Add PHP statements to a watch list so you can see, as you step through the code, how values change.
- Execute additional statements in an immediate window.

Xdebug is a tool that connects your web application (in your local environment) to your code editor. You set breakpoints in your code editor, instruct the editor to begin listening to Debugging Connections, and then refresh the page. Xdebug is easily installed. I have experienced the most difficulty in getting it to properly connect when I am using a virtualized environment.



```
    public function execute(PriceRequestInterface $priceRequest): InternalPriceResponseInterface  $priceRequest: {__phpunit_invocationMocker => PH
    {
        /** @var InternalPriceResponse $response */
        $response = $this->priceResponseFactory->create(['priceRequest' => $priceRequest]);  $this: {calculatorProductFactory => Mock_CalculatorPr

        foreach ($priceRequest->getProducts() as $product) {
            $this->addPrice($product, $priceRequest, $response);
```

\Catalog\PricingEngine\Action › CalculateProductPrice › execute()

```
CalculateProductPrice.php:49, Catalog|PricingEngine|Action|CalculateProductPrice->execute()
CalculateProductPrice.php:42, Catalog|PricingEngine|Test|Unit|Action|CalculateProductPrice->testi
TestCase.php:1069, ReflectionMethod->invokeArgs()
TestCase.php:1069, Catalog|PricingEngine|Test|Unit|Action|CalculateProductPrice->runTest()
TestCase.php:928, Catalog|PricingEngine|Test|Unit|Action|CalculateProductPrice->runBare()
TestResult.php:695, PHPUnit|Framework|TestResult->run()
TestCase.php:883, Catalog|PricingEngine|Test|Unit|Action|CalculateProductPrice->run()
TestSuite.php:744, PHPUnit|Framework|TestSuite->run()
TestSuite.php:744, PHPUnit|Framework|TestSuite->run()
TestRunner.php:537, PHPUnit|TextUI|TestRunner->doRun()
Command.php:212, PHPUnit|TextUI|Command->run()
Command.php:141, PHPUnit|TextUI|Command::main()
phpunit:53, {main}()
```

```
> {...} $priceRequest = {Mock_PriceRequest_cfedd973} [6]
> {...} $this = {Catalog\PricingEngine\Action\CalculateProductPrice} [4]
> $_SERVER = {array} [31]
> $GLOBALS = {array} [46]
> Constants
```

This screenshot is taken from a PHPUnit test case. You can see:

- Where the execution has been stopped by a breakpoint (on line 49).
- The type of each variable (for example, `$priceRequest` is an instance of a PHPUnit test).
- All of the frames to lead to this execution. You can click on a frame to see the execution information at this point.
- The variables for the current frame. When you change the frame, the variables update. You can browse through the variables to understand exactly what is happening at this moment.

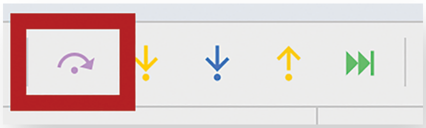
In the companion course for this book (sold separately, read more about it at swiftotter.com/artofdebugging), I walk through some examples of configuring and using Xdebug. I do my best in this written form to convey a lot through screenshots and text, but there's nothing like watching it happen.

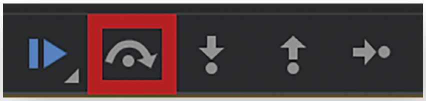
Basic debugger terminology

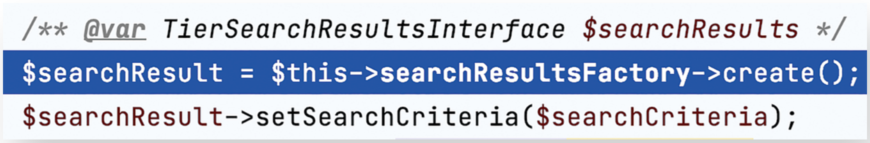
If you have never used a debugger before, you will have a small learning curve – and the nirvana will be the brightest dawn of enlightenment you have ever experienced as a developer.

Breakpoint: This is a marker you place in the code. When the PHP parser arrives at this line, the execution "breaks" and you can begin inspecting variables. Now that the execution has broken and the parser is stopped, you can navigate through code. At any point, you can resume execution (play icon) or stop it (stop icon).

Step Over: In the example below, the parser is on a line that contains another method call. Step Over skips this method call and proceeds to the next line. Think of this as a way to stay in the same method as the previous line of code.

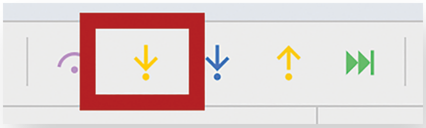

Chrome

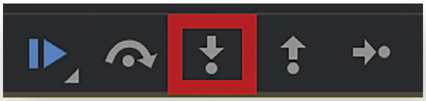

PhpStorm



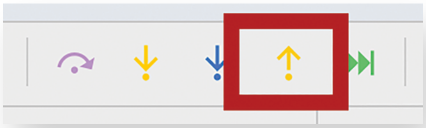
```
/** @var TierSearchResultsInterface $searchResults */
$searchResult = $this->searchResultsFactory->create();
$searchResult->setSearchCriteria($searchCriteria);
```

Step Into: In the above example, the parser will step into this `create` method. By choosing this command, you can inspect the results, line-by-line, of functionality in the `create` method.

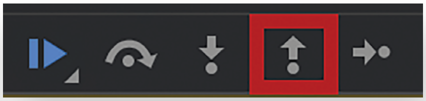

Chrome


PhpStorm

Step Out: This returns to the *next line* of the previous frame. For example, if we stepped into the `create` method, Step Out would take us back and up to `setSearchCriteria`.

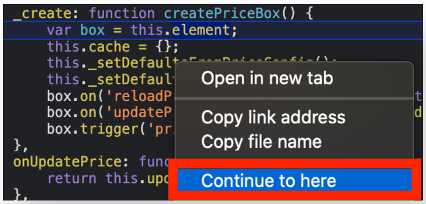


Chrome



PhpStorm

Run to Cursor: The execution will run until it reaches the place where your text cursor is. Think of this as a shortcut to placing a temporary breakpoint in your current debugging session.



```
_create: function createPriceBox() {
    var box = this.element;
    this.cache = {};
    this._setDefault
    this._setDefaul
    box.on('reloadP
    box.on('updateP
    box.trigger('pr
},
onUpdatePrice: func
    return this.upc
},
```

Chrome

Xdebug with a call stack

As we discussed in Chapter 8, call stacks are critical to solving problems. The error itself can be quite obtuse, but the enlightenment happens in the frames before the error is thrown.

Let's review the call stack, and I'll share some thoughts on how to arrive at the locations for breakpoints. Placing too few breakpoints means you'll spend too much time stepping through the code, line-by-line, or you will miss the problematic area. Getting breakpoint-happy and adding too many means you will get lost with too many triggers.

```
NOTICE: ARRAY TO STRING CONVERSION IN /SRV/RELEASES/BUILD-796713766/
VENDOR/LAMINAS/LAMINAS-CRYPT/SRC/UTILS.PHP ON LINE 31

#1 Laminas/Crypt/Utils::compareStrings() called at [vendor/magento/
framework/Encryption/Helper/Security.php:28]
#2 Magento/Framework/Encryption/Helper/Security::compareStrings()
called at [vendor/magento/framework/Data/Form/FormKey/Validator.php:39]
#3 Magento/Framework/Data/Form/FormKey/Validator->validate() called at
[vendor/magento/framework/App/Request/CsrfValidator.php:76]
#4 Magento/Framework/App/Request/CsrfValidator->validateRequest()
called at [vendor/magento/framework/App/Request/CsrfValidator.php:130]
#5 Magento/Framework/App/Request/CsrfValidator->validate() called at
[vendor/magento/framework/App/Request/CompositeValidator.php:40]
#6 Magento/Framework/App/Request/CompositeValidator->validate() called
at [vendor/magento/framework/App/FrontController.php:160]
#7 Magento/Framework/App/FrontController->processRequest() called at
[vendor/magento/framework/App/FrontController.php:118]
#8 Magento/Framework/App/FrontController->dispatch() called at [vendor/
magento/framework/Interception/Interceptor.php:58]
```

(bolded frames will be discussed in the following paragraphs)

Looking up the code in the `Utils` class shows that setting a breakpoint here is worthless – the code could be triggered hundreds of times through the course of a request.

Instead, I would set a breakpoint here:

```
vendor/magento/framework/App/Request/CsrfValidator.php:76
```

This is the closest place to the problem that seems to have functionality specific to the request but doesn't dive into framework utility methods. Even at this point, you might not be able to tell exactly **why** they are set to this value.

You will likely want to set another breakpoint:

```
vendor/magento/framework/App/FrontController.php:118
```

This is the point where the code goes down the rabbit trail of validating the Cross-Site-Request-Forgery token. There is likely something originating here that is passed into the above-mentioned methods that will help us understand what is going on. Why not just start here? Because you need context of the problem – what, specifically, is wrong? Once we have identified this, we can now zoom out for the big picture.

Leave your first breakpoint set as this ensures you have a place to bailout and won't be stuck with continuing to execute your request should you accidentally proceed too far.

Debugger tricks

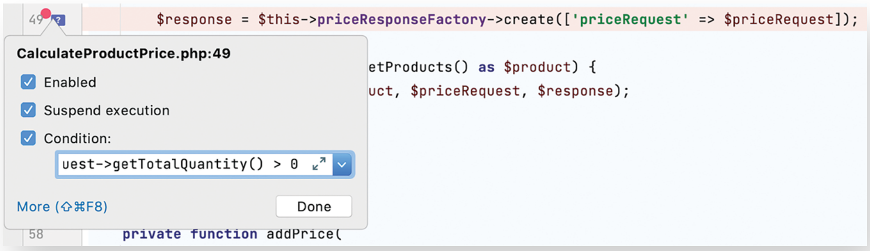
Conditional breakpoints

I'm sure you have needed to set a breakpoint, but that location is called tens of times before the exact moment occurs that you need it.

For example, in the call stack we just reviewed, let's say you need to set a breakpoint in the `compareStrings` method. This method is likely to be called many times in the course of satisfying a web request. We could dutifully step over each of these times – but if you're like me, you will miss that one frame you were trying to catch.

A conditional breakpoint is usually set by right-clicking on the breakpoint and entering a condition. If this expression evaluates to `true`, the breakpoint will trigger. Otherwise, it will not trigger.

Another great example would be where your application is iterating through a long list (XML, a list of customers, or perhaps the list of enabled modules). You can set a breakpoint and add a condition to it where the current value, in the `foreach` loop, equals the customer name.



```
        $response = $this->priceResponseFactory->create(['priceRequest' => $priceRequest]);
etProducts() as $product) {
uct, $priceRequest, $response);
    private function addPrice(
```

CalculateProductPrice.php:49

- ✓ Enabled
- ✓ Suspend execution
- ✓ Condition:

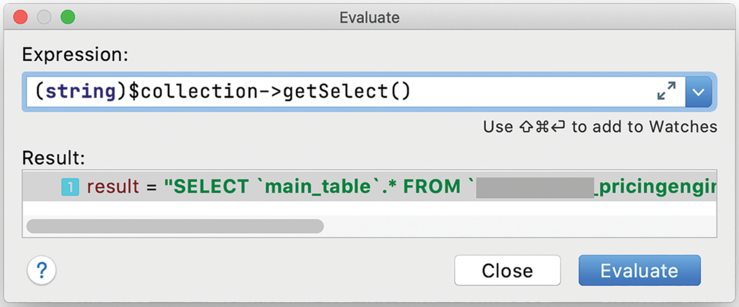
`uest->getTotalQuantity() > 0`

This is a massive time-saver.

Immediate execution

Imagine the power you would feel if you could execute PHP while the parser is paused. For example, what if a class uses the `__toString()` method to provide an important output? In Magento, we cast a SQL statement builder to a string to obtain the resulting SQL query. This means it's impossible to browse the Variables window and understand the final SQL statement.

The Immediate Evaluation tool is just what you need. I use it *all the time.*



```
(string)$collection->getSelect()
```

```
result = "SELECT `main_table`.* FROM `[redacted]_pricingengir
```

Here's a tip for you PHP folks. Don't forget that simply casting a variable to a string can unleash a torrent of code. `__toString()` is a great example. Because this is a method, you could step over this (or trigger this in the Immediate Evaluation window), never realizing this method calls hundreds of lines of code.

Watch expressions

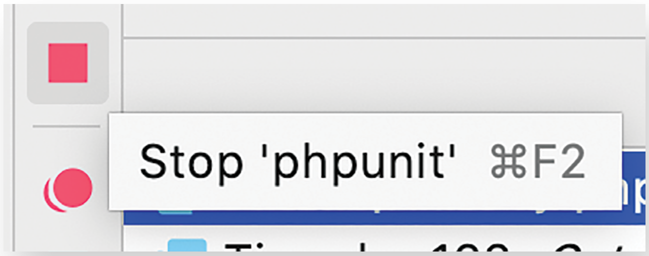
Similar to immediate execution is watch expressions. Once you add a watch expression, it always displays the evaluated result (kind of like an always visible Immediate Expression window). This is helpful if you have to watch multiple values at the same time. I recommend removing watch expressions when you're finished with your debugging to avoid possible issues where a method call in your watch expression could cause an error.

PhpStorm integrates watches into variables. You can right-click on a code selection and select Add to Watches – and this expression will be immediately evaluated (when it is available).

Both Chrome and Firefox have a separate panel for watches.

Stop the execution!

You've probably been in a situation where you are troubleshooting something just before a value is saved to the database (or some other complex operation occurs that is difficult to undo). Instead of letting the parser continue and having to reset the database, simply stop execution:

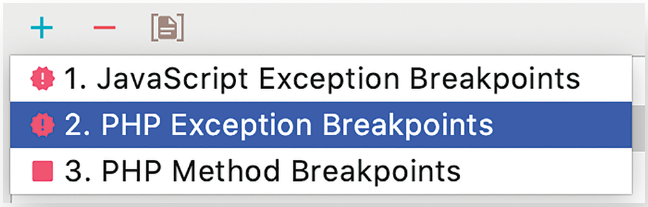


One additional benefit this has on POST requests is that the request is canceled, as you would expect. You can simply refresh the page to trigger the POST request again. This is preferable over having to go through the entire process that led up to the POST request being triggered.

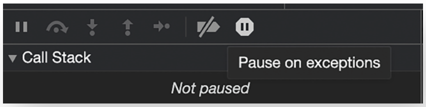
Break when encountering an exception

You can trigger the execution to break when the parser encounters an exception. Wouldn't it be nifty to get in there and understand exactly what is happening?

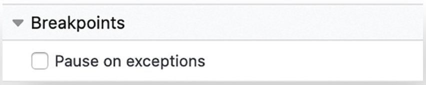
In PhpStorm, you can configure this in Run | View Breakpoints:


1. JavaScript Exception Breakpoints
2. PHP Exception Breakpoints
3. PHP Method Breakpoints

This is also possible in Chrome and Firefox.



Chrome



Firefox

Break before the page unloads

Maybe you backend developers have never experienced this, but ... (I'm guessing you have). Imagine this, you are triggering a Google Analytics event when a link is clicked. You need to use a callback, otherwise the browser kicks in and takes you to your destination. In this process, you have no recourse to understand **why** your code isn't working. Or, better yet, it works *flawlessly* on your local machine but not in production (thus the notorious phrase, "it works for me").

The debugger in Chrome/Firefox Web Developer tools has *the perfect* solution for you. Open the Sources tab, investigate the sidebar (top option is "Watch"). Expand the Event Listener Breakpoints. Expand the Load category and select `beforeunload`. You will notice the myriad of other global breakpoints you can select.

This is a book about debugging, but I can't resist putting a trick in for new development. It can be very handy to write code while the execution is broken. The variables window is populated and some variables have inline type hints. This is like writing code with a heads-up display.

Web Developer tools console

Our discussion of a debugger is not complete unless we work through several additional features of the developer console.

You can open the Web Developer tools with the Ctrl + Shift + I (Windows) or Cmd + Alt + I (Mac). This works with both Chrome and Firefox.

If you are building Vue or React applications, there are plugins available for these frameworks. These will greatly enhance your debugging experience.

Elements tab

Right-click on any element, select Inspect, and you will see this element in the Elements tab. You can change any HTML or CSS.

- These are the elements that build the current page. They may be different from what was rendered in the source code. If there is a difference, you can know with certainty that changes are thanks to JavaScript. Disabling JavaScript for the page will serve as a double-check.

- You can right-click on an element (in the Elements tab) and select "Break On > . ..." When an element or one of its attributes is modified, the debugger will pause allowing you to find "that one thing" that is causing the weird problem on the frontend.

The console

This displays errors (including HTTP errors). In the screenshot below, we see errors relating to Content Security Policies, and this provides the necessary information to make easy updates to your website's CSP.

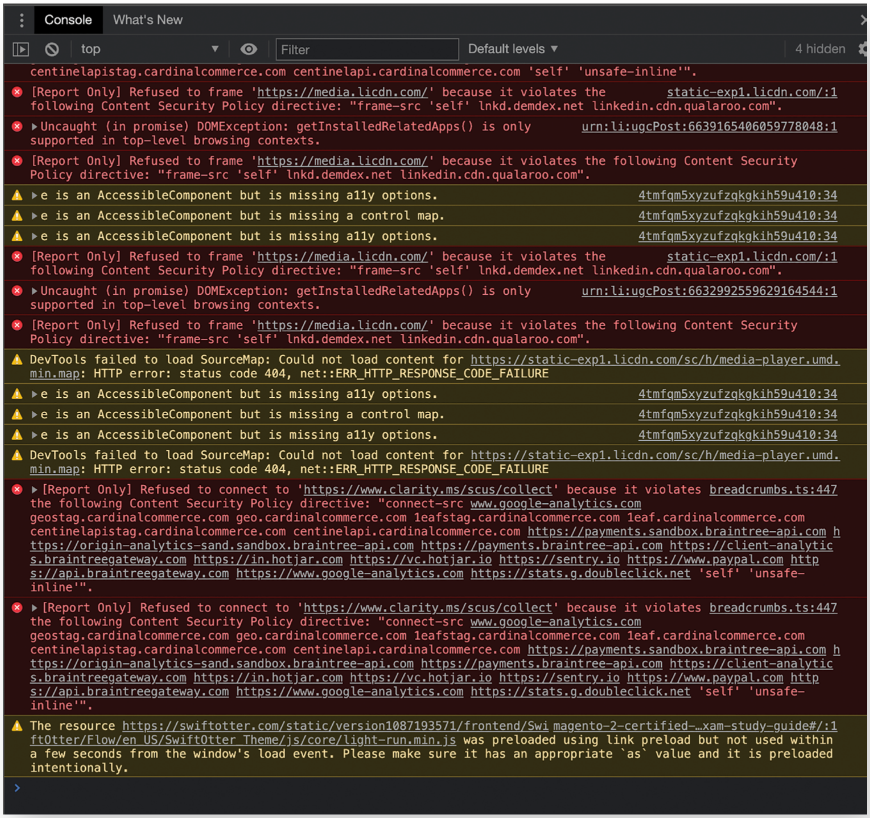
At the bottom of the console is an Immediate Evaluation line. Write whatever text you want here.

Note: You can right-click on a Network request (in the Network tab) and click "Copy as Fetch." Paste the value here. Don't forget to write the response back to the console, such as:

```
fetch("https://mywebsite.com/media/favicon/default/logo-icon.png", {
  "headers": {
    "accept": "image/avif,image/webp,image/apng,image/*,*/*;q=0.8",
    "accept-language": "en-US,en;q=0.9,uk;q=0.8,zh;q=0.7",
    "cache-control": "no-cache",
    "pragma": "no-cache",
    "sec-fetch-dest": "image",
    "sec-fetch-mode": "no-cors",
    "sec-fetch-site": "same-origin"
  },
  "referrer": "https://swiftotter.com/",
  "referrerPolicy": "strict-origin-when-cross-origin",
  "body": null,
  "method": "GET",
  "mode": "cors",
  "credentials": "include"
}).then(response => console.log(response)); // added this then
expression
```

The result is that this becomes somewhat like Postman to send requests – but Postman is much easier to configure. Remember, to copy a request from Chrome and use it in Postman:

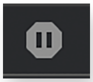
- Right-click on the Network request, and click “Copy as cURL.”
- Go into Postman and click “Import.” Change the type to be Raw Text and paste in the cURL request. Easy.



```
centinelapistag.cardinalcommerce.com centinelapi.cardinalcommerce.com 'self' 'unsafe-inline'".
[Report Only] Refused to frame 'https://media.licdn.com/' because it violates the         static-exp1.licdn.com/:1
following Content Security Policy directive: "frame-src 'self' lnkd.demdex.net linkedin.cdn.qualaroo.com".
▸Uncaught (in promise) DOMException: getInstalledRelatedApps() is only      urn:li:ugcPost:6639165406059778048:1
supported in top-level browsing contexts.
[Report Only] Refused to frame 'https://media.licdn.com/' because it violates the following Content Security
Policy directive: "frame-src 'self' lnkd.demdex.net linkedin.cdn.qualaroo.com".
▸e is an AccessibleComponent but is missing a11y options.                    4tmfqm5xyzufzqkgkih59u410:34
▸e is an AccessibleComponent but is missing a control map.                   4tmfqm5xyzufzqkgkih59u410:34
▸e is an AccessibleComponent but is missing a11y options.                    4tmfqm5xyzufzqkgkih59u410:34
[Report Only] Refused to frame 'https://media.licdn.com/' because it violates the         static-exp1.licdn.com/:1
following Content Security Policy directive: "frame-src 'self' lnkd.demdex.net linkedin.cdn.qualaroo.com".
▸Uncaught (in promise) DOMException: getInstalledRelatedApps() is only      urn:li:ugcPost:6632992559629164544:1
supported in top-level browsing contexts.
[Report Only] Refused to frame 'https://media.licdn.com/' because it violates the following Content Security
Policy directive: "frame-src 'self' lnkd.demdex.net linkedin.cdn.qualaroo.com".
DevTools failed to load SourceMap: Could not load content for https://static-exp1.licdn.com/sc/h/media-player.umd.
min.map: HTTP error: status code 404, net::ERR_HTTP_RESPONSE_CODE_FAILURE
▸e is an AccessibleComponent but is missing a11y options.                    4tmfqm5xyzufzqkgkih59u410:34
▸e is an AccessibleComponent but is missing a control map.                   4tmfqm5xyzufzqkgkih59u410:34
▸e is an AccessibleComponent but is missing a11y options.                    4tmfqm5xyzufzqkgkih59u410:34
DevTools failed to load SourceMap: Could not load content for https://static-exp1.licdn.com/sc/h/media-player.umd.
min.map: HTTP error: status code 404, net::ERR_HTTP_RESPONSE_CODE_FAILURE
▸[Report Only] Refused to connect to 'https://www.clarity.ms/scus/collect' because it violates breadcrumbs.ts:447
the following Content Security Policy directive: "connect-src www.google-analytics.com
geostag.cardinalcommerce.com geo.cardinalcommerce.com 1eafstag.cardinalcommerce.com 1eaf.cardinalcommerce.com
centinelapistag.cardinalcommerce.com centinelapi.cardinalcommerce.com https://payments.sandbox.braintree-api.com h
ttps://origin-analytics-sand.sandbox.braintree-api.com https://payments.braintree-api.com https://client-analytic
s.braintreegateway.com https://in.hotjar.com https://vc.hotjar.io https://sentry.io https://www.paypal.com http
s://api.braintreegateway.com https://www.google-analytics.com https://stats.g.doubleclick.net 'self' 'unsafe-
inline'".
▸[Report Only] Refused to connect to 'https://www.clarity.ms/scus/collect' because it violates breadcrumbs.ts:447
the following Content Security Policy directive: "connect-src www.google-analytics.com
geostag.cardinalcommerce.com geo.cardinalcommerce.com 1eafstag.cardinalcommerce.com 1eaf.cardinalcommerce.com
centinelapistag.cardinalcommerce.com centinelapi.cardinalcommerce.com https://payments.sandbox.braintree-api.com h
ttps://origin-analytics-sand.sandbox.braintree-api.com https://payments.braintree-api.com https://client-analytic
s.braintreegateway.com https://in.hotjar.com https://vc.hotjar.io https://sentry.io https://www.paypal.com http
s://api.braintreegateway.com https://www.google-analytics.com https://stats.g.doubleclick.net 'self' 'unsafe-
inline'".
The resource https://swiftotter.com/static/version1087193571/frontend/Swi magento-2-certified-…xam-study-guide#/:1
ftOtter/Flow/en_US/SwiftOtter_Theme/js/core/light-run.min.js was preloaded using link preload but not used within
a few seconds from the window's load event. Please make sure it has an appropriate `as` value and it is preloaded
intentionally.
>
```

Pause on exception

Developer tools also have the ability to stop when an exception is triggered. Toggle this button "on" and the debugger will pause when the next exception is thrown. When enabled, you can check "Pause on caught exceptions" – especially helpful to find those silenced errors.



Over 25% of you reading this book do not use Xdebug (according to a February 2021 survey that I created). I'm guessing that for those of you who do use Xdebug, there are probably a few additional tricks you could pick up.

Network panel

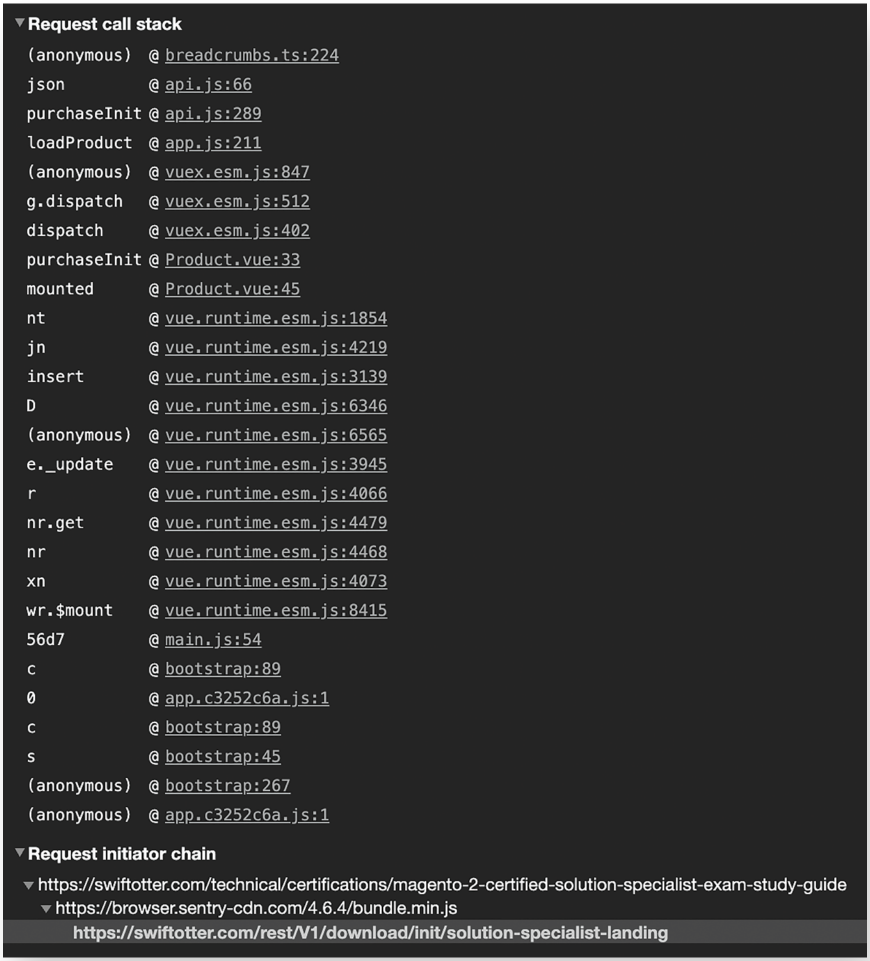
Here are a few of my favorite tricks for this tab:

- Requests that are triggered via XHR (the “old” way) are able to be retriggered. Right-click on the request and click “Replay XHR.” Requests triggered by `fetch` do not have this capability – although you can right-click on the request and click “Copy as Fetch.” As we just discussed in the section titled “The console,” I document how to get the results of these `fetch` requests.

- “Copy as cURL” allows you to “export” requests into your `bash` command line (or to Postman).

- To slow down requests, you can simulate a slow connection. At the top of the Network Inspector, there is a drop-down menu for request speed. The default is “Online.” You can select from Fast 3G or Slow 3G. These slow methods help you get a better picture of load order. Remember how we discussed those “random” JavaScript load problems? Simulating a slow network just might be the tool to land you on a solution.

But what JavaScript triggered this network request? Only recently did I uncover the trick for this: Select a request in the Network panel and click on the Initiator tab. You will see *exactly* what triggered this request:


```
▼ Request call stack
(anonymous)  @ breadcrumbs.ts:224
json         @ api.js:66
purchaseInit @ api.js:289
loadProduct  @ app.js:211
(anonymous)  @ vuex.esm.js:847
g.dispatch   @ vuex.esm.js:512
dispatch     @ vuex.esm.js:402
purchaseInit @ Product.vue:33
mounted      @ Product.vue:45
nt           @ vue.runtime.esm.js:1854
jn           @ vue.runtime.esm.js:4219
insert       @ vue.runtime.esm.js:3139
D            @ vue.runtime.esm.js:6346
(anonymous)  @ vue.runtime.esm.js:6565
e._update    @ vue.runtime.esm.js:3945
r            @ vue.runtime.esm.js:4066
nr.get       @ vue.runtime.esm.js:4479
nr           @ vue.runtime.esm.js:4468
xn           @ vue.runtime.esm.js:4073
wr.$mount    @ vue.runtime.esm.js:8415
56d7         @ main.js:54
c            @ bootstrap:89
0            @ app.c3252c6a.js:1
c            @ bootstrap:89
s            @ bootstrap:45
(anonymous)  @ bootstrap:267
(anonymous)  @ app.c3252c6a.js:1
▼ Request initiator chain
 ▼ https://swiftotter.com/technical/certifications/magento-2-certified-solution-specialist-exam-study-guide
   ▼ https://browser.sentry-cdn.com/4.6.4/bundle.min.js
       https://swiftotter.com/rest/V1/download/init/solution-specialist-landing
```

Use the HTML source to zero in on the problem area

When you are trying to determine what code triggers a particular set of functionality, reviewing the source code to find commonalities is helpful.

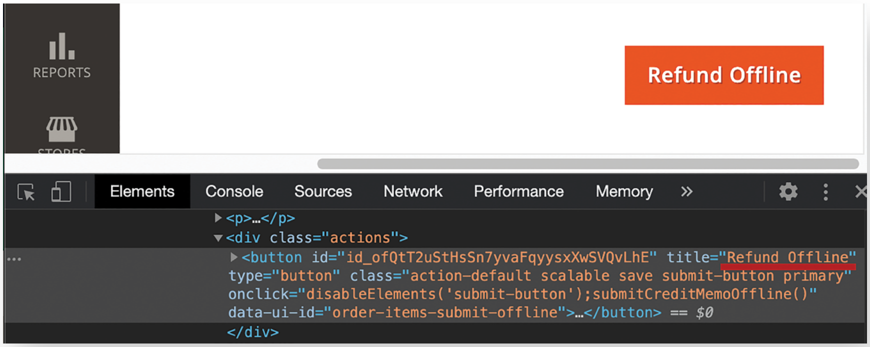
Remember the example I shared back in the Take inventory section – about how credit memos weren't working?

This is the tool I used to triangulate where I should start investigating:

- I navigated to the offending screen to create a credit memo and saw the "Refund" button was missing.

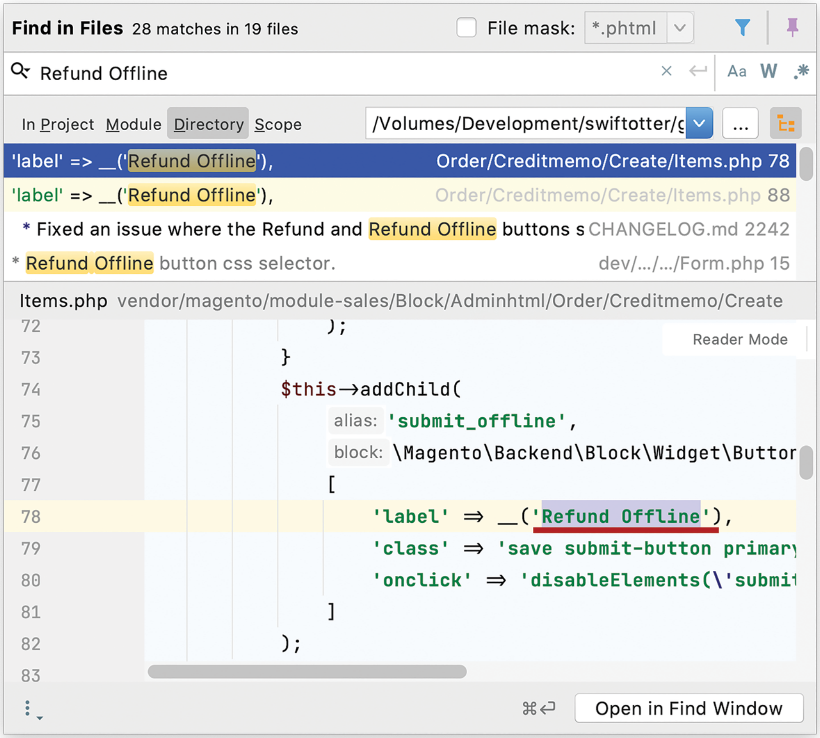
- I opened up Chrome Developer tools and used the Inspector button to highlight Refund Offline.

- I looked for a CSS class or something that was unique to this button. The `title` attribute or the `data-ui-id` attribute would qualify:



```
<p>…</p>
<div class="actions">
  <button id="id_ofQtT2uStHsSn7yvaFqyysxXwSVQvLhE" title="Refund Offline"
  type="button" class="action-default scalable save submit-button primary"
  onclick="disableElements('submit-button');submitCreditMemoOffline()"
  data-ui-id="order-items-submit-offline">…</button> == $0
</div>
```

- I took the value of the `title` attribute, `Refund Offline`, and searched through the codebase:



Find in Files 28 matches in 19 files
File mask: *.phtml
Refund Offline
Aa W .*
In Project Module Directory Scope
/Volumes/Development/swiftotter/g
Reader Mode
⌘↵ Open in Find Window

```
'label' => __('Refund Offline'),                               Order/Creditmemo/Create/Items.php 78
'label' => __('Refund Offline'),                               Order/Creditmemo/Create/Items.php 88
 * Fixed an issue where the Refund and Refund Offline buttons s CHANGELOG.md 2242
* Refund Offline button css selector.                          dev/.../.../Form.php 15
```

Items.php vendor/magento/module-sales/Block/Adminhtml/Order/Creditmemo/Create

```
            );
        }
        $this→addChild(
            alias: 'submit_offline',
            block: \Magento\Backend\Block\Widget\Buttor
            [
                'label' ⇒ __('Refund Offline'),
                'class' ⇒ 'save submit-button primary
                'onclick' ⇒ 'disableElements(\'submit
            ]
        );
```

- I scrolled up a few lines and found this `if` statement:

```
if ($this→getCreditmemo()→canRefund()) {
```

- I now had a jumping off point. It turns out the missing Refund button was only a symptom of another more difficult-to-troubleshoot problem.

Once you find the location, I suggest setting a breakpoint and getting it to trigger. This will tell you if you found the right spot. Depending on the snippet you searched for, and the size of your platform, you might have found *a* location but not *the* location. This could save you significant time.

Performance problems

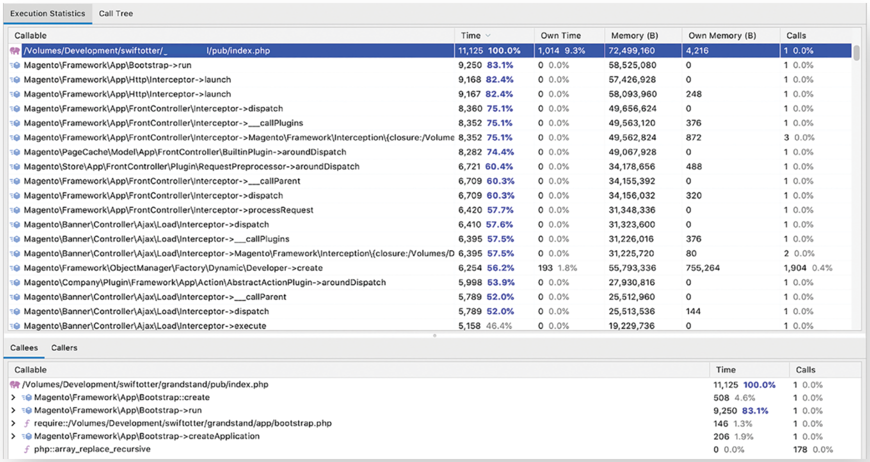
Performance problems can be a serious challenge. I hope the number you have to face is few and far between, but it's inevitable you will face performance optimizations. I laid out some methodologies in Chapter 8 for how to work through these.

The first place to check is your performance monitoring tool (New Relic or Tideways, for example). This will give you an excellent perspective into what is happening. If you have Blackfire installed and configured, use this to locate performance issues.

The next *tool* you should use is the profiler for your environment (Xdebug for PHP or the Web Developer Tools profiler in your browser). Xdebug provides KCacheGrind-compatible outputs that PhpStorm can parse.

To get a profile for a given request, set the `XDEBUG_MODE=profile` environment variable or change the `xdebug.mode` value in your PHP configuration file. You should also configure the `xdebug.output_dir` in your PHP configuration file to point to the directory with the logging points you wish to observe.

You can go into PhpStorm and select Tools > Analyze Xdebug Profiler Snapshot. Here is the result:


Execution Statistics | Call Tree

Callable	Time	Own Time	Memory (B)	Own Memory (B)	Calls
/Volumes/Development/swiftotter/_ l/pub/index.php	11,125 100.0%	1,014 9.3%	72,499,160	4,216	1 0.0%
Magento\Framework\App\Bootstrap->run	9,250 83.1%	0 0.0%	58,525,080	0	1 0.0%
Magento\Framework\App\Http\Interceptor->launch	9,168 82.4%	0 0.0%	57,426,928	0	1 0.0%
Magento\Framework\App\Http\Interceptor->launch	9,167 82.4%	0 0.0%	58,093,960	248	1 0.0%
Magento\Framework\App\FrontController\Interceptor->dispatch	8,360 75.1%	0 0.0%	49,656,624	0	1 0.0%
Magento\Framework\App\FrontController\Interceptor->__callPlugins	8,352 75.1%	0 0.0%	49,563,120	376	1 0.0%
Magento\Framework\App\FrontController\Interceptor->Magento\Framework\Interception\{closure:/Volume	8,352 75.1%	0 0.0%	49,562,824	872	3 0.0%
Magento\PageCache\Model\App\FrontController\BuiltinPlugin->aroundDispatch	8,282 74.4%	0 0.0%	49,067,928	0	1 0.0%
Magento\Store\App\FrontController\Plugin\RequestPreprocessor->aroundDispatch	6,721 60.4%	0 0.0%	34,178,656	488	1 0.0%
Magento\Framework\App\FrontController\Interceptor->__callParent	6,709 60.3%	0 0.0%	34,155,392	0	1 0.0%
Magento\Framework\App\FrontController\Interceptor->dispatch	6,709 60.3%	0 0.0%	34,156,032	320	1 0.0%
Magento\Framework\App\FrontController\Interceptor->processRequest	6,420 57.7%	0 0.0%	31,348,336	0	1 0.0%
Magento\Banner\Controller\Ajax\Load\Interceptor->dispatch	6,410 57.6%	0 0.0%	31,323,600	0	1 0.0%
Magento\Banner\Controller\Ajax\Load\Interceptor->__callPlugins	6,395 57.5%	0 0.0%	31,226,016	376	1 0.0%
Magento\Banner\Controller\Ajax\Load\Interceptor->Magento\Framework\Interception\{closure:/Volumes/D	6,395 57.5%	0 0.0%	31,225,720	80	2 0.0%
Magento\Framework\ObjectManager\Factory\Dynamic\Developer->create	6,254 56.2%	193 1.8%	55,793,336	755,264	1,904 0.4%
Magento\Company\Plugin\Framework\App\Action\AbstractActionPlugin->aroundDispatch	5,998 53.9%	0 0.0%	27,930,816	0	1 0.0%
Magento\Banner\Controller\Ajax\Load\Interceptor->__callParent	5,789 52.0%	0 0.0%	25,512,960	0	1 0.0%
Magento\Banner\Controller\Ajax\Load\Interceptor->dispatch	5,789 52.0%	0 0.0%	25,513,536	144	1 0.0%
Magento\Banner\Controller\Ajax\Load\Interceptor->execute	5,158 46.4%	0 0.0%	19,229,736	0	1 0.0%

Callees | Callers

Callable	Time	Calls
/Volumes/Development/swiftotter/grandstand/pub/index.php	11,125 100.0%	1 0.0%
> Magento\Framework\App\Bootstrap::create	508 4.6%	1 0.0%
> Magento\Framework\App\Bootstrap->run	9,250 83.1%	1 0.0%
> require::/Volumes/Development/swiftotter/grandstand/app/bootstrap.php	146 1.3%	1 0.0%
> Magento\Framework\App\Bootstrap->createApplication	206 1.9%	1 0.0%
php::array_replace_recursive	0 0.0%	178 0.0%

Within this window, you can:

- Sort by “Own Time” – how long this particular method took.
- Sort by “Own Memory” – how much memory this method used.
- Sort by “Calls” – how many times this method was called.

The strategies presented above will get you a long way to identifying pain points in your application.

Chapter 10

DEBUGGING TOOLS

Say, what's with the rubber duck?
Whenever I encounter a coding problem, I explain the code line by line to the duck—and that way, I usually discover the issue, so—
That seems kind of silly. Why not just talk to a real person instead?
The rubber duck doesn't interrupt.

It sounds ironic to say this, but before I fully jumped into “computers,” my career was headed toward construction. I think this is why I still enjoy purchasing and using tools. In fact, remodeling is somewhat of a hobby for me. I am quick to reach for tools to help me do a better job.

The tools I discuss in this next section are ideas that have enhanced my debugging capabilities. They aren’t as concrete as using the debugger. But they have helped me solve many problems.

The module upgrade

Yes, that title just stole my thunder. Please continue reading.

I was recently assigned a task that a couple of our other developers couldn’t solve. It was a tough problem: The Magento site used a module to integrate WordPress. The Elementor module is installed in WordPress. In other words, this is a house of cards.

You guessed it; Elementor was broken. We always got an error when trying to preview a page in Elementor. I tried turning on logging. Nothing showed up.

We turned to Elementor support. They couldn’t help much because of the Magento integration. We were at a dead-end. What to do?

I reviewed the modules involved in this integration and found that the Magento modules were out of date. I upgraded them and held my breath … would it work?!

Whalla! It did! That was a *huge* relief.

When you are dealing with a significant problem with other modules, always check to see if an update exists.

The rubber duck

Have you ever solved a problem by simply talking it through? You know, you are venting to someone about this or that, and it’s like you reach into your pocket and find a silver bullet! The solution *somehow* becomes obvious. You’ve been wracking your brain to determine the solution for hours ... and it just appears *as you are talking it through!* Maybe magic is real, after all?!

I can’t tell you how many times this has occurred to me – or someone who was talking to me.

There is actually a Wikipedia article called “Rubber duck debugging.” Talking through problems provides solutions because we are thinking out loud. There is something that seems to click in our brains when we put words with our thoughts.

The reason a rubber duck was the original idea is that a piece of plastic is inanimate. It won’t talk back or provide suggestions before you are through with your idea. It goes without saying if you are functioning as a rubber duck for someone else’s problem, do them a favor by listening to their entire discourse before jumping in and providing a solution.

Next time you are having trouble while in the *Take inventory* step, hop on a Zoom call with a friend or step into their office. Explain your problem. You just might have a revelation.

“Find the most knowledgeable person and ask them.”

I put this in quotes because this is not a genuine recommendation. This is one of your best ways to torpedo your hopes of being a great developer.

Here is the scenario I have observed: You run into a difficult issue, *and the first thing you do is go talk with a senior developer.* You do no research. You build no case. And, of course, you have no plan of action. This is a great way to annoy people – so save yourself the pain and don’t do it.

Instead, work through the TAD (take inventory, attempt a fix, do it again) framework at least once. If this doesn’t provide a solution, you will have information you can present to your senior developer.

If you are unsure whether or not your solution is the right one, please take your ideas to your senior developer and ask her opinion. Remember, in doing this, you are bringing an informed solution to the senior developer – and not simply asking her to do your job for you.

Draw it out

I give credit on this one to my dad (an electrical engineer – so think “schematics”). I am the type who likes to dig in and get to work. I have found, however, that investing the time up front to build a drawing can be quite beneficial. This is especially helpful when I am traversing a deep valley of debugging. When you have a ton of components working together and are having trouble making sense of their relationships, *build a diagram.*

A search engine

Come on, man. Seriously, you think we *don't know how to use a search engine?* I agree, this seems elementary. But you would be surprised how many developers rely on "who knows what" to find the solution to a problem. For example, if there is a word in this book you don't know, search for it (that said, I did include a dictionary in the Appendix).

A search engine is a starting place for research. Otherwise, paralysis results (and I've seen it a ton of times). The developer doesn't know what to do or where to go. The result is ... nothing happens. No solutions and no progress but many hours spent. I sure know this feeling – and you probably do too.

Google (or DuckDuckGo for a privacy-centric alternative) is a great resource for finding solutions. Here are my favorite tips:

- Always include the name of your technology in the search box. For example (no punctuation or capitalization is required – that takes extra time to type):

```
magento 2 transaction id not set on invoice
laravel class user not found
```

  Notice that I put the version number in the search string, if it is important/applicable.
- You can usually search for an error message. You still want to put the platform name in the search string.

```
magento 2 Broken reference: The element with the "category.view.
container" ID wasn't found. Verify the ID and try again.
```

- Wrap quotes around strings to require an exact match:

```
Magento 2 “Invalid Form Key. Please refresh the page.”
```

- You can search a specific site with:

```
site:stackexchange.com Laravel ReflectionException
```

Using a search engine does not mean you are a bad developer. You are likely saving hours of time. How long does it take to do a search? 15 seconds or less? Do the search quickly – if you don’t find anything, abort the process, and go back to your debugging.

Temporary code modifications

We’ve been strictly warned *never* to modify core code (yes, thank you, Ben Marks). That’s right, we do the unpardonable, the hammer falls, and we little peons are squashed into ... who knows what.

However, I’ve found *temporary* code modifications to be an incredibly effective tool to facilitate fast solutions.

For example, in Magento, once an order is placed, you are redirected to the “order success” page. The code to render this page also clears the session (i.e., the cart). If you needed a value in that session, you must go back through the process to create an order.

Or, you can locate the code in the success controller that cleans the session and comment it out – temporarily. In these cases, you should ask yourself "Why is this happening?" and "What can I do to make my life easier?"

If you are editing code in a package-managed directory, like the `vendor` folder, you may want to add a consistent comment above your core code modifications so you can easily find all core edits in the future. Something like:

```
// core edit
<your code edits>
```

While you can reset your package-management directory by deleting its contents and running the install command again, I've often found it useful to be able to review all my core edits by searching for a consistent string.

But what about those pieces that were built by that clueless previous developer?

You inherit a project where the client is hoping to start with new developers and a fresh relationship. Your salespeople are confident your company can right this ship, so they put their very best developer on the project (you). You get into the code and the only words that keep surfacing in your mind are 4-letter, NSFW words.

Debugging these projects is difficult – but you likely have a weapon the previous developer did not have (if the code is really bad): the debugger. The other thing I do is slowly reformat and rebuild one piece at a time. You've heard the saying, "How do you eat an elephant? One bite at a time." Pieces are more palatable to the merchant than having to swallow the cost of a replatform.

Logging

Intelligently-placed log entries can be your biggest lifesaver.

If you need to debug on-the-fly in production (uh oh!), logging allows you to keep the website working while getting intelligent information written to files on the backend. Now with the PSR-3 standard, you can utilize many logging solutions, all with the same interface: `LoggerInterface`.

Logging is especially helpful when you work with a read-only production environment – and you are unable to make any changes to the code.

What should I log?

- Any and all exceptions should be logged, unless the exception is expected and is part of the flow of logic.
- Include all pertinent information. PSR-3 compliant methods include two parameters:
  `$message`: The message is the primary text written to the logs. But you have an even better tool (see the next point).
  `$context`: The context is an array. You can add whatever details you wish to add in this place. The more the merrier. This is usually where I log the stack trace from exceptions that are thrown.
- Feel free to add logging into core libraries. You can do this in a semi future-proof way with Composer patches (cweagans/composer-patches).

How should I log the things?

- PHP has a significant number of libraries that are PSR-3 compliant. These allow you to log to files, or anywhere, really.
- If you need immediate notification of a problem, *and the problem doesn't happen very often,* consider sending yourself an email. This is useful for those “random” problems where I need more information.

For example, we maintain a custom Customer Relationship Manager (CRM) for one of our merchants. This merchant reported a problem in which saving a quote would occasionally zero out the total. We couldn't turn up anything in our investigation. A zero-dollar quote total is unhelpful to the prospect. We added email logging and waited for a replication. Sure enough, it came. We had the information necessary to resolve the problem. We were able to immediately notify the merchant of this zero-dollar quote along with what they could do to prevent this from happening (then we built a permanent fix for this issue).

- The frontend is more difficult in that there is no easy way to communicate logs back to yourself. We have used several tools that are lightweight and do a great job:

    - FullStory: This is a system that "records" the visitor's monitor. It also tracks JavaScript errors. This is a fairly ideal solution in that you can easily see all the steps taken before the error. You literally have a full replication in front of your eyes. All errors are saved – real-time – in their DevTools panel.

    - Sentry.io: We use this tool on all of our frontend applications. The specific service doesn't matter as much as your having a robust way to understand when errors happen and *why they happen.* These tools usually integrate with Slack so you can get immediate feedback when visitors encounter an error.

      We also use this to aggregate our backend application logs and are able to track those errors easily. There are many other options for this; if your team has a devops expert, ask him how to aggregate logs.

- Establish a log management program. Consider using `logrotate`. This Unix-based program will enforce limits on log files. Old files are at least compressed. Depending on the value and size of the logs, you could consider deleting old ones. Evaluate carefully before deleting a file as you will invariably need the one file you just deleted. Log files are the window into the past state of an application.

  For example, one of our merchants came to us with a serious problem: Almost 1,000 of their products were deleted. This is a situation where my heart seems to inhabit my throat. Talk about stress? Yep, that's it. I'm not just worried about getting the data back. I have to know *why* this happened.

  I spent some time examining the HTTP access log files and found there was a POST request to the delete product URL – from the IP this merchant uses. It was damning evidence – just sayin'.

- Triangulating log files. I ran into a situation where some records were updated in the database. There was a specific workflow that would trigger this update, but the log file time-stamps proved this workflow wasn't being triggered. I reviewed the Nginx access logs, searching for the time when this record was incorrectly updated. I found the API was guilty of updating this record.

How should I parse log files?

Log files present little value unless you are comfortable extracting information from them. We will be just scratching the surface in presenting rudimentary techniques on parsing log files. There are many SaaS offerings and tools that all make this process easier – if you are willing to learn them.

- `tail` is a Unix program that fetches the last X number of lines from a file. You can make it follow the output with the `-f` switch (like `tail -f [filename]`). You will see, real-time, all additions to this log file.

  Conversely, `head` returns the first lines from a file.

- `grep` is a Unix program that finds matching lines within a file. This is a powerful way to filter out values in log files. You can combine the two of these with bash using the `|` operator.

  The `r` flag is recursive. This will look through the current direction and all sub directories.

  The `n` flag prints out the line number of matches. This allows you to find the location quickly.

  The `w` flag matches specific words. Let's look at this string: `the quick brown fox`. Specifying the `w` flag, `grep -w "brown"` would match the word `brown`. `grep -w "q"` would not provide a match.

  Here are some examples:

  - `tail -f access.log | grep "192\.123\.191\.12"`: This finds all entries associated with an IP address. We are using the `-f` flag so we see all new additions to this access log.

  - `grep -rnlw "[TEXT HERE]". | sort | xargs ls -al`: This is useful for finding matches in a large number of files (i.e., not just one log file). This looks for a text match, sorts by the dates the files were written, and writes out the filename. You now know the files that contain this string.

You can also watch multiple log files at once:

```
tail -fn0 \
    /home/example/example.com/current/var/log/*.log \
    /home/example/var/php-fpm/error.log  \
    /var/log/nginx/example.com/logs/error.log  \
    /var/log/nginx/example.com/logs/error-ssl.log
```

Erik Hansen shared this tip. He puts this command in every project he works with, and it allows him to quickly identify all important paths. Brilliant!

- I just learned about `zgrep` (thank you, Silvia). This is a tool similar to `grep` but *it searches compressed files* (`.gz` , for example). In the past, I used to unzip the file, then `grep` it – but not anymore!

- Don't forget the humble `vim` text editor! At some point in your career, you will be forced to use this – i.e., when you debug staging or production.

  I estimate I know less than 5% of its functionality, yet I feel like I can get around pretty well! You will do yourself a favor by spending even one hour watching videos and learning how to use it.

  `vim` can do everything a normal text editor can do and more.

  `:set nowrap` is one of the more useful vim commands when reviewing log files, as it turns off line wrapping, making logs much easier to read.

A huge gotcha I've stumbled over several times is timestamps. Remember that correctly configured logs are entered in the GMT time zone. Unless you are one of those fine chaps who are lucky enough to live in this narrow slice of our world, you have to convert times. When you come across that entry in the log files, *make sure it is the right date and time zone.*

Yes, and that is before you go share the snippet with your client, saying you found the problem. They might write back saying, "You got the wrong date." Glug.

Personal solution log

I give credit for this suggestion to Lee Saferite.

Remember at the beginning of this book I stated the two components that make a senior developer? The first is time on the job, which translates to problems solved. The more time you spend in Magento, the more commonalities you quickly identify.

There is one massive exception: our memory. You might be one of the few great folks who truly have an outstanding memory. I'll be the first to say I don't fit into that boat.

This can be implemented very simply: Create a Google Doc or a GitHub Gist. When you encounter an interesting solution, write it down. As you add information into this document, you will have an indescribable treasure trove. Years and years of this discipline will exponentially increase your value as a developer.

Yes, I’ve started mine and have already logged a few entries. I intend to continue adding to it.

In conclusion

You have now read my tips, tricks, and secrets for debugging ecommerce applications. While I have learned them over the course of approximately 10 years, these ideas are yours, today.

We have discussed how it is worthless to read a book and not take its suggestions. You are better than that. Take one tool at a time and learn how to use it. Use it as often as possible. You will thus become successful.

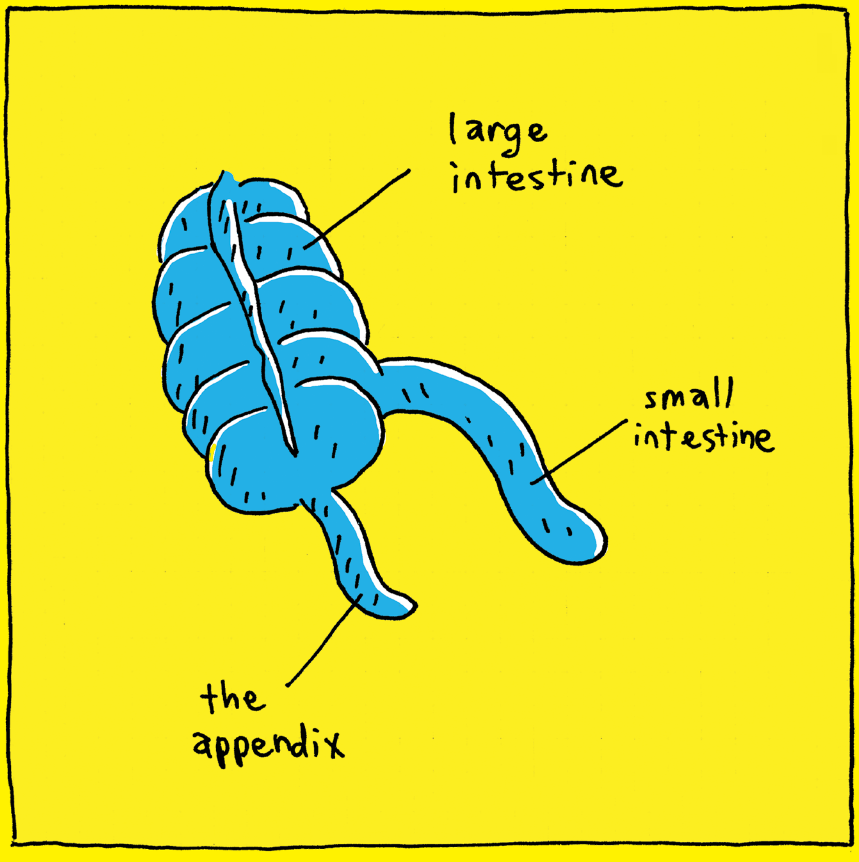
I wish you many happy years of effectively solving problems. It would be my greatest pleasure to have a part in helping you achieve your goals in life through your becoming a better and faster debugger.

For more resources

I’m compiling (and keeping up-to-date) a list of helpful resources and references on our website: swiftotter.com/artofdebugging

You can also sign up for information – so you can stay at the top of your debugging game.

APPENDIX



Definitions

API (application programming interface)

A set of functions and procedures allowing the creation of applications that access the features or data of an operating system, application, or other service. (Oxford Languages)

APIs typically return structured data (JSON, XML, GraphQL). When one system needs information from another system, it is often best to use an API. API requests usually load faster than HTML requests because the API call only has data associated with fulfilling the requests and doesn't contain images or HTML elements.

Nginx / Apache

These are systems that accept web requests and return a response. They can send the request to PHP or another language processor.

Linux

An operating system that powers almost every ecommerce-enabled website. It is Unix-based.

Bash / ZSH

The name of the command line that operates on Linux. These offer different feature sets but the same basic functionality.

VIM

The *best* text editor for Linux. It doesn't rely on a mouse – instead, all navigation happens with keyboard shortcuts.

ElasticSearch
The name of a search system. It stores records and indexes them and returns search results very quickly.

RabbitMQ
A message queuing system. Messages are sent to RabbitMQ and RabbitMQ forwards them on to a particular process. This is very useful for asynchronous programming to speed up web requests by offloading expensive processes.

Frameworks, like Symfony and Laravel
These are code bases that contain the structure to build and deploy applications. They contain code to load information from the web request. They have a truckload of utilities that streamline development – but none of them is specifically geared toward a particular purpose.

Platforms, such as Magento and WooCommerce
These are typically very large codebases that may or may not be built on a framework. Platforms are intended to accomplish one purpose, in this case, facilitating transactions online. One of your best sources of information is the documentation found on their website.

The Art of Ecommerce Debugging

Course for Magento

If you enjoyed this book and you work with Magento, may I recommend the companion course?

My goal with this book is to provide practical suggestions that apply to almost every ecommerce platform. To this end, I have generalized some of my suggestions to fit within a broader scope of platforms.

In my course, however, I have the opportunity to focus explicitly on Magento and share even more of what I have learned.

What's in the course?

The course contains real-world bugs that we get to troubleshoot *together*. You get a database and codebase. We have a bug report, and we work through the problem, step-by-step, with a series of videos (and written instructions). It's like sitting next to your senior developer, except this isn't a live website and you are under no time pressure to complete the task. You get hands-on, real-life experience – but it doesn't come with the price tag of working on a merchant's site.

You will also get access to our exclusive Slack community where others are learning debugging with Magento (this is different from the Slack community that is generally dedicated to this book).

You can read more about the course at:
swiftotter.com/artofdebugging

I look forward to introducing companion courses for other specific platforms—Shopify, BigCommerce, etc.
Sign up for updates on swiftotter.com.