

O'REILLY®

DevOps Tools for Java Developers

Best Practices from Source Code
to Production Containers

Early
Release

RAW &
UNEDITED



Stephen Chin, Melissa McKay,
Ixchel Ruiz & Baruch Sadogursky

DevOps Tools for Java Developers

Best Practices from Source Code to Production
Containers

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

**Stephen Chin, Melissa McKay, Ixchel Ruiz, and
Baruch Sadogursky**



DevOps Tools for Java Developers

by Stephen Chin, Melissa McKay, Ixchel Ruiz, and Baruch Sadogursky

Copyright © 2022 Stephen Chin, Baruch Sadogursky, Melissa McKay, and Ixchel Ruiz. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://oreilly.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Acquisitions Editor: Suzanne McQuade

Development Editor: Corbin Collins

Production Editor: Kate Galloway

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: O'Reilly Media

February 2022: First Edition

Revision History for the Early Release

- 2020-12-18: First Release
- 2020-01-07: Second Release
- 2021-06-11: Third Release
- 2021-09-01: Fourth Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781492084020> for release

details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *DevOps Tools for Java Developers*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the authors, and do not represent the publisher's views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-492-08395-5

Chapter 1. DevOps for (or Possibly Against) Developers

Baruch Sadogursky

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors’ raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book. If there is a GitHub repo associated with the book, it will be made active after final publication.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at ccollins@oreilly.com.

“While you here do snoring lie, Open-eyed conspiracy His time doth take. If of life you keep a care, Shake off slumber, and beware: Awake, awake!”

—William Shakespeare, *The Tempest*

Some might ask if the DevOps movement is simply an Ops-inspired plot against developers? Most (if not all) who’d do so wouldn’t expect a serious response, not least because they intend the question as tongue-in-cheek teasing. It’s also because – and regardless if your origins are on the Dev or the Ops side of the equation – when anyone strikes up a conversation about DevOps, it will take approximately 60 seconds before someone inquires: “But what DevOps *really* is?”

And you’d think, ten years after the coining of the term (a decade within which industry professionals have spoken, debated, and shouted about it), that we’d all have arrived at a standard, no-nonsense, commonly-understood definition. But this simply isn’t the case. In fact, despite an exponentially-increasing corporate demand for DevOps personnel, it’s highly doubtful that any five DevOps-titled

employees, chosen at random, could tell you *precisely* what DevOps is. So, don't be embarrassed if you still find yourself scratching your head when the subject comes up. Conceptually, DevOps may not be easy to grok, but it's not impossible either.

But regardless of how we discuss the term or what definition(s) we might agree upon, there's one thing, above all else that's critical to bear in mind:

DevOps is an entirely invented concept, and the inventors came from the Ops side of the equation

That may be provocative, but it's provable, too. Let's start with two exhibits.

Exhibit #1: The Phoenix Project

Co-written by three info-tech professionals, *The Phoenix Project, A Novel About IT, DevOps, and Helping Your Business Win* ¹ was released in 2013. It's not a how-to manual (not in the traditional sense, anyway). It's a novel that tells the story of a highly problematic company and its IT manager who is suddenly assigned the task of implementing a make-or-break corporate initiative that's already way over budget and months behind schedule. If you dwell in the realms of software, the rest of the book's central characters will be very familiar to you. For the moment, though, let's have a look at their professional titles:

- Director, IT Service Support
- Director, Distributed Technology
- Manager, Retail Sales
- Lead Systems Administrator
- Chief Information Security Officer
- Chief Financial Officer
- Chief Executive Officer

Notice the connective tissue between them? They're the protagonists of one *the* most important books about DevOps ever written and *not one* of them is a developer. Even when developers do figure into the plotline, well...let's just say they're not spoken of in particularly glowing terms. When victory comes, it's the hero of the story (together with a supportive board member) who invents DevOps, pulls the project's fat out of the fire, turns his company's fortunes around, and gets rewarded with a promotion to CIO of the enterprise. And everyone lives happily – if not ever after, then for at least the two or three years such successes tend to buy you in this business before it's time to prove your worth all over again.

Exhibit #2: The DevOps Handbook

It's better to read *The Phoenix Project* before *The DevOps Handbook, How to Create World-Class Agility, Reliability, & Security in Technology Organizations*² because the former places you within a highly believable, human scenario. It's not difficult to immerse yourself in the personality types, the professional predicaments, and the interpersonal relationships of the characters. The hows and whys of DevOps unfold as inevitable and rational responses to a set of circumstances, which could have just as easily led to business collapse. The stakes, the characters, and the choices they make all seem quite plausible. Parallels to your own experience may not be too hard to draw.

The DevOps Handbook allows you to explore the component conceptual parts of DevOps principles and practices in greater depth. As its subtitle suggests, the book goes a long way toward explaining “How to Create World-Class Agility, Reliability, and Security in Technology Organizations.” But shouldn't that be about development? Whether it should or shouldn't may be open to debate. What's incontrovertible is the fact that the book's authors are bright, super-talented professionals who are, arguably, the fathers of DevOps. However, Exhibit #2 isn't included here to praise them so much as to take a close look at their backgrounds.

Let's start with **Gene Kim**. He founded the software security and data integrity firm, Tripwire, for which he served as its Chief Technology Officer for over a decade. As a researcher, he's devoted his professional attentions to examining and understanding the technological transformations that have and are occurring

within large, complex businesses and institutions. He also co-authored *The Phoenix Project* and, in 2019, *The Unicorn Project* (about which, more later). Everything about his career is steeped in Ops. Even when *Unicorn* says it's "about Developers," it's still developers as seen through the eyes of an Ops guy!

As for the other three authors of the *Handbook*:

- **Jez Humble** has held positions including Site Reliability Engineer (SRE), CTO, Deputy Director of Delivery Architecture and Infrastructure Services, and Developer Relations. An Ops guy! Even where the last of his titles references development, the job isn't *about* that. It's about relations *with* developers. It's about narrowing the divide between Dev and Ops, about which he has written, taught, and lectured extensively.
- **Patrick Debois** has served as a CTO, Director of Market Strategy, and Director of Dev♥Ops Relations (the heart is his addition). He self-describes himself as a professional who is "bridging the gap between projects and operations by using Agile techniques in development, project management, and system administration." That sure sounds like an Ops guy.
- **John Willis**, as of this writing, holds the title of VP of DevOps and Digital Practices. Previously, he's been a Director of Ecosystem Development, a VP of Solutions, and notably a VP of Training and Services at Opscode (now known as Chef). And while John's career has been a bit more deeply involved with development, most of his work has been about Ops, particularly to the degree that he has focused his attentions on tearing down the walls that had once kept developers and operations personnel in very distinct and separate camps.

As you can see, all the authors have an Ops background. *Coincidence? I think NOT.*

Still not convinced that DevOps is Ops driven? How about we have a look at who are the leaders trying to sell us on DevOps today.

Google It

In mid-2020, if you entered the term “What is DevOps?” in a Google search just to see what would come up, it’s likely that your first page of results would have included the following:

- Agile Admin, a system administration company
- Atlassian, whose products include project and issue tracking, list making, and team collaboration platforms
- AWS, Microsoft Azure, and Rackspace, all of which sell Cloud Ops infrastructure
- Logz.io, which sells log management and analysis services
- New Relic, whose specialty is application monitoring

All of these are very Ops-focused. Yes, that first page contained one firm that was a bit more on the development side and one other which really wasn’t directly related to the search. The point here is that when you try to look for DevOps, most of what you’ll find tends to skew towards Ops.

What Does It Do?

DevOps *is* a thing! It’s in *big* demand. And with this, there are many who will want to know, concretely, what DevOps *does*, what it substantively produces. Rather than go down that route, let’s look at it structurally, conceptualizing it as you would the sideways, figure 8-shaped infinity symbol. In this light, we see a loop of processes that goes from coding to building to testing to release to deployment to operations to monitoring, and then back again to begin planning for new features.



Figure 1-1. DevOps Infinity Loop

And if this looks familiar to some readers, it should because it bears a conceptual similarity to the Agile development cycle:

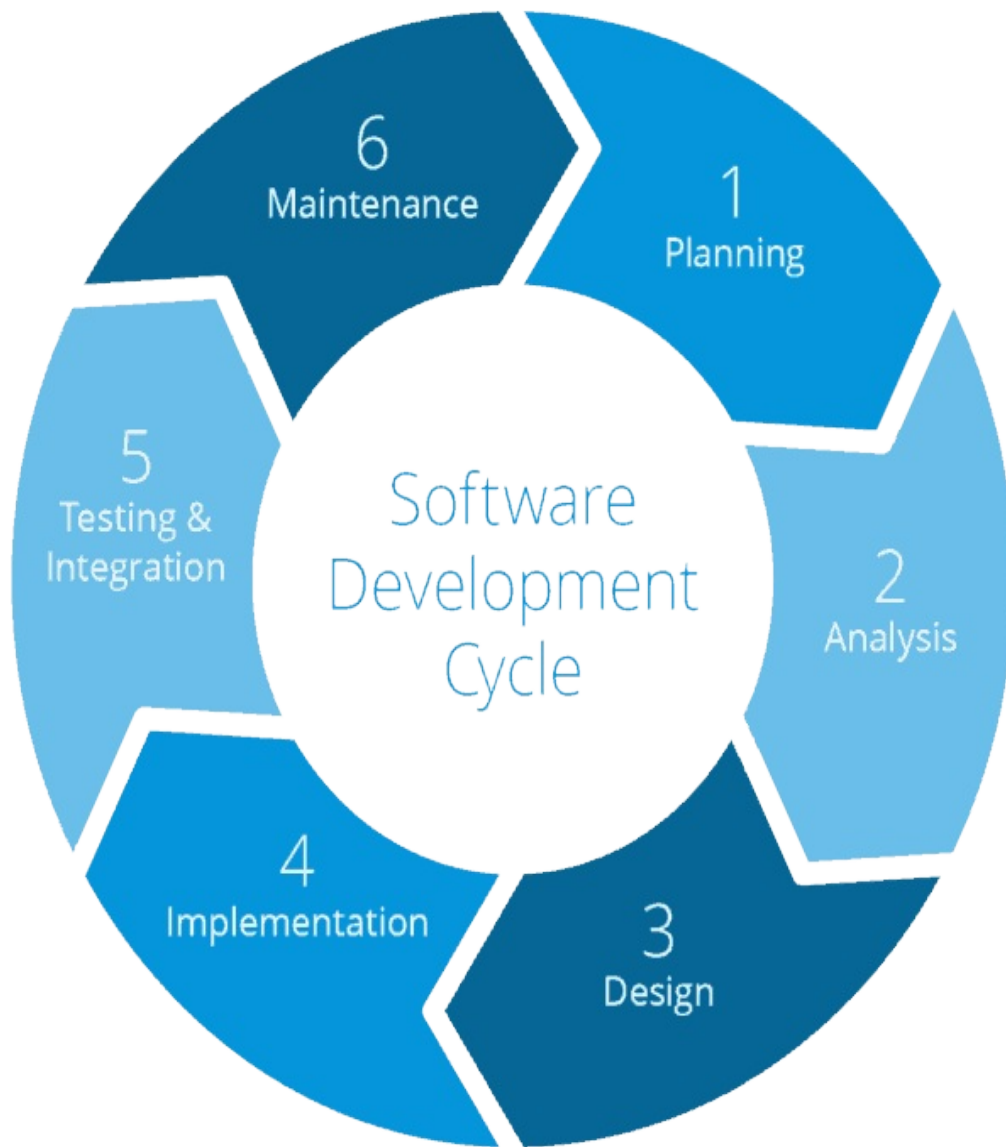


Figure 1-2. Agile Development Cycle

There's no profound difference between these two, never-ending stories other than the fact that Ops folks grafted themselves onto the old world of the Agile circle, essentially stretching it across two circles and checkmarking their concerns

circle, essentially stretching it across two circles and shoehorning their concerns and pains into the domain that was once considered solely the province of developers.

State of the Industry

There's further proof that DevOps is an Ops-driven phenomenon which, since 2014, has come packaged in the form of an easy-to-read, annual digest of data collected, analyzed, and summarized from tens of thousands of industry professionals and organizations worldwide. The Accelerate State of DevOps Report was primarily the work of DevOps Research and Assessment (DORA) and it's *the* most important document in the software industry to gauge where DevOps is and where it's likely going. In the 2018 edition ³, for example, we can see a serious focus on questions such as:

- How often do organizations deploy code?
- How long does it typically take to go from code commits to successfully running in production?
- When impairments or outages occur, how long does it generally take to restore service?
- What percentage of deployed changes result in degraded service or require remediation?

Notice that all of those are very Ops-centered concerns.

What Constitutes Work?

Now let's examine how *work* is defined by the *State of DevOps Report* and in *The Phoenix Project*. Well first, there's planned work that focuses on business projects and new features, which spans both Ops and Dev. Internal projects, including server migrations, software updates, and changes driven by feedback on deployed projects can be broad-based and may or may not weight more to one side of the DevOps equation than the other. But what about unplanned activities, such as support escalations and emergency outages. Those are very Ops heavy, as is the coding of new features, bug fixes, and refactoring, which are all about how the life of Ops can be made easier by *including* developers in

the DevOps story.

If We're Not About Deployment and Operations, Then Just What /s Our Job?

So, it's quite clear that DevOps isn't anything developers were (or are) demanding. It's an Ops invention to make everyone else work harder. And assuming this truth, let's ponder what would happen if developers were to rise up as one and say, "Your Ops concerns are *yours*, not ours." Fine. But in that case it would be only right and proper to ask the revolting developers what is their definition of "done." What standards do they believe they need to achieve to say, "We did our job well and our part is now complete"?

This isn't a flippant question and there are sources we can look to for answers. One, although imperfect and not infrequently criticized, is the *Manifesto for Software Craftsmanship* ⁴, which puts forth four fundamental values that should motivate developers. Let's consider them:

- "Well-crafted software." Yes, indeed, quality is important.
- "Steadily adding value." No argument there. Of course we want to provide services and features that people need, want, or would like.
- "Community of professionals." In broad strokes, who could oppose this? Cordiality among industry peers is just being professionally neighborly.
- "Productive partnerships." Collaboration is certainly the name of the game. Developers aren't against QA, Ops, or the products themselves. So, in context, this is just about being friendly with everybody (so long as other teams don't start dictating what their jobs are supposed to be).

Just What *Does* Constitute Done?

With all that we've established so far, we can safely say that we need to produce code that's simple, readable, understandable, and easy to deploy. We must be certain that non-functional requirements (e.g., performance, throughput, memory footprint) are satisfied. We should work diligently to avoid incurring any

technical baggage and, if we're lucky, shedding some along the way. We have to be sure that all tests get passed. And we're obligated to maintain fruitful relations with QA teams (when they're happy, we're happy).

With good quality code, plus positive team leader and peer reviews, everything should be fine out of the gate. With a product team that defines standards for value and added value, benchmarks can be firmly established. Through their feedback, product owners help to determine if those benchmarks are (or aren't) being met and by how much. That's a pretty good thumbnail definition of a good software developer having "done" what they needed to do. It also demonstrates that well done can never adequately be measured (or even known) without the involvement of and clear communications with Ops personnel.

Rivalry?

So yes, although it *can* be proven that DevOps really wasn't anything for which developers were clamoring, it can equally be shown how its infinite practice benefits everyone. And still there are holdouts; those who imagine that there's a rivalry, even an antagonism between developers and, for example, QA testers. Developers work hard on their creations and then feel like the QA teams are almost like hackers, out to prove something, just digging and digging until they find a problem.

This is where DevOps counseling comes in. Every conscientious developer wants to be proud of what they're producing. Finding flaws can seem like criticism, when it's really just conscientious work coming from another direction. Good, clear, open, and continuous conversations between developers and QA personnel helps to reinforce the benefits of DevOps, but it also makes plain that *everyone* is ultimately working toward the same goal. When QA folks identify bugs, all they're doing is helping their developer colleagues to write better code, to be *better* developers. And this example of interplay between those on the Ops side and others on the Dev side demonstrates the useful blurring of the distinctions and separations between the two worlds. Their relationship is necessarily symbiotic and, once again, works along an endless continuum of activities with one informing the other for the mutual benefit of all.

More Than Ever Before

The increasing demand for DevOps is coming as much from external forces as it is from within software companies themselves. And this is because our expectations, *all* of our expectations, as people living in a 21st century world, continue to change rapidly. The more reliant we become on ever-improving software solutions, the less time there is to waste on information and communication gaps, and delays between developers and operations personnel.

Take, for example, banking. A decade ago, most major banks had reasonably adequate websites. You could log in, have a look at your accounts, your statements and recent transactions, maybe you were even beginning to make payments electronically through the e-services your bank was offering. And while it was nice and offered a certain amount of convenience, it's likely you still needed (or, at least, felt more comfortable) going to your local branch to handle your banking affairs.

What didn't exist is today's fully digital experience, complete with mobile apps, automated account monitoring and alerts, and enough services that make it more and more common for average account holders to do *everything* online. You may even be among those who not only don't care if you ever see the inside of your bricks-and-mortar branch ever again, you don't even *know* where that branch is! What's more, banks are reacting to these rapidly changing banking habits by consolidating and closing physical branches, and offering incentives for their customers to shift their banking to the online sphere. This accelerated even more during the COVID-19 crisis when access to branches was being restricted to appointment-only services, limited walk-in access, and shorter hours.

So, whereas ten years ago, if your bank's website went down for twelve hours of maintenance while they were deploying a better, more secure site, you'd probably have taken that in stride. What's a dozen hours if it's going to result in higher quality services? You didn't need 24/7, online banking and, besides, the local branch was there for you. Today, that's simply not the case. Half a day's worth of downtime is unacceptable. In essence, you expect your bank to *always* be open and available for you. This is because your (and the world's) definition of quality has changed. And that change requires DevOps more than ever before.

Volume and Velocity

-

Another pressure impelling the growth of DevOps is the amount of data that's being stored and handled. And that's only logical. If more and more of our daily lives are reliant on software, then there's obviously going to be a tremendous rise in that realm. In 2020, the entire, global datasphere amounted to nearly 10 zettabytes. A decade prior it was 0.5 zettabytes. By 2025, it's reasonably estimated that it will balloon exponentially to over 50 zettabytes!⁵

This isn't only about behemoths like Google, Netflix, Twitter, and others getting bigger and better, and therefore needing to process larger amounts of data. This projection affirms that more and more companies will be graduating into the world of Big Data. With that comes the demands of vastly increased data loads, as well as the shift away from traditional staging server environments, which offered exact replicas of given production environments. And this shift is predicated on the fact that maintaining such pair-to-pair schemes is no longer feasible in terms of size or speed.

Happy were the days of yore when everything could be tested before going into production, but this isn't possible anymore. Things are and will increasingly be released into production about which software firms don't have 100% confidence. Should this cause us to panic? No. The necessity to release fast and remain competitive should inspire innovation and creativity in the ways required to best execute controlled rollovers, test procedures, and more *in-production* testing – what's now referred to as *progressive delivery*. This comes along with feature flags and observability tools, such as distributed tracing.

There are some who equate progressive delivery with the blast radius of an explosive device. The idea is that when one deploys to an in-production environment, explosions should be expected. Therefore, to optimize such rollouts, the best one can hope for is to minimize casualties, to keep the size of the blast radius as small as is possible. This is consistently accomplished through improvements in the quality of servers, services, and products. And if we agree that quality is a concern of developers and its achievement is part of a developer's definition of done, then it means there can be no pause or disconnect between that Dev moment of done and the next, Ops moment of production. Because no sooner than this happens then we're looping back into development, as bugs are fixed, services restored due to outages, and so on.

Done and Done

Maybe it's becoming clear that it's the expectations and demands that were and continue to be generated from the Ops environment, which, of necessity, drove the push to DevOps. And as a result, the increased expectations and demands on developers isn't coming from some festering hatred Ops folks have for their developer colleagues, nor is it part of a plot to deprive them of sleep. Rather, *all* of this, all of what DevOps represents is a realpolitik business response to our changing world and the changes they've forced on the software industry across the board.

The fact is that everyone has new responsibilities, some of which require professionals (certainly many departments) to be at the ready to respond *whenever* duty calls because ours is a non-stop world. Another way of putting this is: Our old definitions of done are done! Our new definition is SRE, or Site Reliability Engineering. It's a Google-coined term that forever weds Dev to Ops by bridging any lingering, perceived gaps between the two. And while SRE areas of focus may be taken up by personnel on either or both sides of the DevOps equation, these days companies will often have dedicated SRE teams that are specifically responsible for examining issues related to performance, efficiency, emergency responsiveness, monitoring, capacity planning, and more. SRE professionals think like software engineers in devising strategies and solutions for system administration issues. They're the folks who are increasingly making automated deployments work.

When SRE staff are happy, it means builds are becoming ever more reliable, repeatable, and fast, particularly because the landscape is one of scalable, backwards and forwards compatible code operating in stateless environments that are relying on an exploding universe of servers and emitting event streams to allow for real-time observability and alerts when something goes wrong. When there are new builds, they need to launch rapidly (with the full expectation that some will die equally quickly). Services need to return to full functionality as rapidly as possible. When features don't work, there must be an immediate ability to turn them off programmatically through an API. When new software is released and users update their clients, but then encounter bugs, there must be the ability to execute swift and seamless rollbacks. Old clients and old servers need to be able to communicate with new clients. And while SRE is assessing and monitoring these activities and laying out strategic responses to them. the

work in all of these areas is completely that of developers.

Therefore, while Dev staff are doing, it's SRE that is today's definition of done.

Fly Like a Butterfly...

In addition to all of the considerations above, there's a fundamental characteristic that must define code in our modern DevOps (and related SRE) era: lean. And by this we're talking about saving money. "But what," you may ask, "does code have to do with saving money?" Well, one illustration might be cloud providers who will charge companies for a plethora of discrete services. Some of this is directly affected by the code being output by those corporate cloud service subscribers. Cost reductions can therefore come from the creation and use of innovative developer tools, as well as writing and deploying better code.

The very nature of a global, we-never-close, software-driven society whose constant desire for newer, better software features and services means that DevOps can't only be concerned with production and deployments. It *must* also be attentive to the bottom line of business itself. And although this may seem to be yet another burden thrown into the mix, think about it the next time the boss says that costs must be cut. Instead of negative, kneejerk solutions, such as employee layoffs or reductions in salaries and benefits, needed savings may be found in positive, business profile enhancing moves like going serverless and moving to the cloud. Then, no one gets fired and the coffee and donuts in the breakroom are still free!

Being lean not only saves money, it gives companies the opportunity to improve their marketplace impact. When firms can achieve efficiencies without staff reductions, they can retain optimal levels of team strength. When teams continue to be well compensated and cared for, they'll be more motivated to produce their best possible work. When that output is achieving success, it means customers are grinning. So long as customers continue to get well-functioning, new features quickly as a result of faster deployments, well... they'll keep coming back for more and spreading the word to others. And *that's* a virtuous cycle that means money in the bank.

Integrity Authentication and Availability

Integrity, Authentication, and Availability

Hand in hand, and shoulder to shoulder with any and all DevOps activities is the perpetual question of security. Of course, some will choose to address this concern by hiring a Chief Information Security Officer. And that's great because there'll always be a go-to person to blame when something goes wrong. A better option might be to actually analyze, *within* a DevOps framework, how individual employees, work teams, and companies-as-a-whole *think* about security and how it can be strengthened. We are going to talk much more about this in Chapter 11, but for now consider the following:

Breaches, bugs, SQL injections, buffer overflows, and more aren't new. What's different is the increasing speed of their appearance, their quantity, and the cleverness with which malicious individuals and entities are able to act. It's not surprising. With more and more code being released, more and more problems will follow, with each type demanding different solutions.

With faster deployments it becomes ever more critical to be less reactive to risks and threats. The discovery, in 2018, of the Meltdown and Spectre security vulnerabilities made it clear that some threats are impossible to prevent. We are in a race, and the only thing to do is to deploy fixes as quickly as possible.

Fierce Urgency

It should be clear to the reader by now that DevOps is not a plot, but a response to evolutionary pressures. It's a means to ends that:

- Deliver better quality
- Produce savings
- Deploy features faster
- Strengthen security

And it doesn't matter who likes it or not, it doesn't matter who came up with the idea first, it doesn't even matter what was its original intent. What matters is that

The software industry has fully embraced DevOps

By now, every company is a DevOps company.⁶ So, get on board... because you don't have any other choice.

The DevOps of today, what DevOps has evolved into is, as stated earlier, an infinity loop. It doesn't mean that groups and departments no longer exist. It doesn't mean that everyone is responsible for their areas of concern along with those of everyone else along this continuum. It *does* mean that everyone should be working together. It *does* mean that software professionals within a given enterprise must be aware of and taking into reasonable consideration what all of their other colleagues are doing. They need to care about the issues their peers are confronting, how those matters can and do impact the work *they* do, the products and services their company offers, and how the *totality* of this affects their firm's marketplace reputation.

This is why DevOps Engineer is a term that makes no sense because it implies the existence of someone who can comprehensively and competently do (or is, at least, completely versed in) everything that occurs within the DevOps infinity loop. There is no such person. There never will be. In fact, even *trying* to do so is a mistake because it runs entirely counter to what DevOps is, which is eliminating silos where code developers are walled off from QA testers who are walled off from release personnel, and so on.

It's a coming together of efforts, interests, and feedback in a continuous effort to create, secure, deploy, and perfect code. DevOps is about *collaboration*. And as collaborations are organic, communicative endeavors, well... just as collaboration engineering isn't a thing, neither is DevOps engineering (no matter what any institute or university might promise).

Making It Manifest

Knowing what DevOps is (and isn't) only establishes a concept. The question is how can it be sensibly and effectively implemented and sustained in software firms far and wide? Best advice? Here goes:

- You can have DevOps enablers, DevOps evangelists, DevOps consultants and coaches (and I know how Scrum spoiled all those terms for all of us, but there aren't any better ones). That's okay. But DevOps

is *not* an engineering discipline. We want site/service reliability engineers, production engineers, infrastructure engineers, QA engineers, and so on. But once a company has a DevOps Engineer, the next thing it's almost guaranteed to have is a DevOps Department, which will just be another silo that's as likely as not to be nothing more than an existing department that's been rebranded, so it *looks* like the firm is on the DevOps bandwagon. A DevOps office isn't a sign of progress. Rather, it's simply back to the future. Then, the next thing that will be needed is a way to foster collaborations between Dev and DevOps, which will necessitate the coining of yet another term. How about *DevDevOps*?

- It's about nuances and small things. Like cultures (especially corporate cultures), it's about attitudes and relationships. You may not be able to clearly define those cultures, but they exist all the same. It's also not about code, engineering practices, or technical prowess. There's no tool you can buy, no step-by-step manual, or home edition board game that will help you to create DevOps in your organization. It's about *behaviors* that are encouraged and nurtured within companies. And much of this is simply about how rank-and-file staff are treated, the structure of the firm, and the titles people hold. It's about how often people have an opportunity to get together (especially in non-meeting settings), where they sit and eat, talk shop and non-shop, tell jokes, etc. It's not within data centers, but the spaces within which cultures form, grow, and change.
- Companies should actively seek and invest in T-shaped people (Ж-shaped is even better, as my Russian-speaking readers might suggest). As opposed to I-shaped individuals (who are absolute experts in one area) or generalists (who know a good bit about a lot, but have no mastery of any particular discipline), a T-shaped person has world-class expertise in at least one thing. This is the long vertical line of the "T" that firmly roots their depth of knowledge and experience. The "T" is crossed above by a breadth of accumulated capabilities, know-how, and wisdom in other arenas. The total package is someone who demonstrates a clear and keen propensity to adapt to circumstances, learn new skills, and meet the challenges of today. In fact, this is a

nearly perfect definition of an ideal DevOps staffer. T-shaped personnel allow businesses to work effectively on prioritized workloads, instead of only what companies think their in-house capacities can bear. T-folks can see and are intrigued by the big picture. This makes them terrific collaborators which, as a result, leads to the construction of empowered teams.

We all got the message

The good news is that a decade after Ops invented DevOps, they completely understand that it's not only about them. It's about *everybody*. We can see the change with our own eye. For example, the 2019 State of DevOps Report got **more developers to participate in the study, than Ops or SRE personnel**⁷. To find another profound evidence that things have changed, we return, full circle to Gene Kim. Also in 2019, the man who helped to novelize the Ops end of the equation with *The Phoenix Project* released *The Unicorn Project: A Novel About Developers, Digital Disruption, and Thriving in the Age of Data*⁸. If the earlier book gave short shrift to developers, here our hero is Maxine, her company's **lead developer** (and ultimate savior).

DevOps began with Ops, no question about it. But the motivation wasn't the subjugation of developers, nor the supremacy of operations professionals. It was and remains predicated on everyone seeing everyone else, appreciating their value and contributions to an enterprise – not simply out of respect or courtesy, but as personal self-interest and business survival, competitiveness, and growth.

And if you're scared that DevOps is going to overwhelm you in a sea of Ops concepts, it's actually most likely to be the other way around. Just look at SRE's definition by Google (the company which invented the discipline)⁹:

SRE is what you get when you treat operations as if it's a software problem.

So, the ops folks want to be developers now? Welcome. Software problems belong to *all* software professionals *all* the time. We're in the problem-solving business. Which means that everyone is a little bit of an SRE, a little bit of a developer, a little bit into operations...because it's all the same thing. They're all just intertwined facets that allow us to devise solutions for the software of today,

as well as the individual and societal problems of tomorrow.

- 1 The Phoenix Project, A Novel About IT, DevOps, and Helping Your Business Win (<https://itrevolution.com/the-phoenix-project/>)
- 2 The DevOps Handbook, How to Create World-Class Agility, Reliability, & Security in Technology Organizations (<https://itrevolution.com/book/the-devops-handbook/>)
- 3 DORA State of DevOps Report 2018 (<https://inthecloud.withgoogle.com/state-of-devops-18/dl-cd.html>)
- 4 Manifesto for Software Craftsmanship (<https://manifesto.softwarecraftsmanship.org/>)
- 5 Annual size of real time data in the global datasphere from 2010 to 2025 (<https://www.statista.com/statistics/949144/worldwide-global-datasphere-real-time-data-annual-size/>)
- 6 By 2020 Every Company will be a DevOps Company (<https://jfrog.com/blog/by-2020-every-company-will-be-a-devops-company/>)
- 7 The 2019 Accelerate State of DevOps Report, Demographics and Firmographics (<https://cloud.google.com/devops/state-of-devops>)
- 8 The Unicorn Project A Novel about Developers, Digital Disruption, and Thriving in the Age of Data (<https://itrevolution.com/the-unicorn-project/>)
- 9 What is Site Reliability Engineering (SRE)? (<https://sre.google/>)

Chapter 2. The System of Truth

Stephen Chin

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors’ raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book. If there is a GitHub repo associated with the book, it will be made active after final publication.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at ccollins@oreilly.com.

A complex system that works is invariably found to have evolved from a simple system that worked.

—John Gall (Gall’s Law)

To have an effective DevOps pipeline, it is important to have a single system of truth to understand what bits and bytes are being deployed into production. For most systems this starts with a source code management system that contains all of the source code that gets compiled and built into the production deployment. By tracing a production deployment back to a specific revision in source control, you can do root cause analysis of bugs, security holes, and performance issues.

Source code management solves several key roles in the software delivery lifecycle:

1. *Collaboration:* Large teams working on a single codebase would constantly get blocked by each other without effective source code management, reducing productivity as the team size grows.
2. *Versioning:* Source code systems let you track different versions of the code to identify what is being deployed into production or released to a

customer.

3. *History*: By keeping a chronological record of all versions of software as it is developed, it is possible to revert back to an older version of the code or identify the specific change that caused a regression.
4. *Attribution*: Knowing who made the changes in a particular file allows you to identify ownership, domain expertise, and assess risk when making changes.
5. *Dependencies*: Source code has become the canonical source for other key metadata about the project like dependencies on other packages.
6. *Quality*: A source code management system allows for easy peer review of changes before they are accepted, increasing the overall quality of the software.

Since source code management plays such a critical role in software development, it is important to understand how it works and select a system that best meets the needs of your organization and the desired DevOps workflow.

Three Generations of Source Code Management

Collaboration is a big part of software development, and as you scale with larger teams, the ability to collaborate effectively on a shared code base often becomes a bottleneck to developer productivity. Also, the complexity of systems tends to increase, so rather than managing a dozen files or a handful of modules, it is common to see thousands of source files that need to be updated en masse to accomplish system-wide changes and refactorings.

To manage the need to collaborate on codebases, source code management (SCM) systems were created. The first generation SCM systems handled collaboration via file locking. Examples of these are SCCS and RCS, which required that you lock files before editing, make your changes, and then release the lock for other folks to contribute. This seemingly eliminated the possibility of two developers making conflicting changes with two major drawbacks:

1. Productivity was still impacted since you had to wait for other developers to finish their changes before editing. In systems with large

files, this could effectively limit the concurrency to only one developer at a time.

2. This did not solve the problem of conflicts across files. It is still possible for two developers to modify different files with inter-dependencies and create a buggy or unstable system by introducing conflicting changes.

A substantial improvement over this was made in the second generation version control systems starting with Concurrent Versions System (CVS) created by Dick Grune. CVS was revolutionary in its approach to (or lack of) file locking. Rather than preventing you from changing files it would allow multiple developers to make their simultaneous (and possibly conflicting) changes to the same files. This was later resolved via file merging where the conflicting files were analyzed via a difference (diff) algorithm and any conflicting changes were presented to the user to resolve.

By delaying the resolution of conflicting changes to a “check-in”, CVS allowed multiple developers to freely modify and refactor a large codebase without becoming blocked on other changes to the same files. This not only increases developer productivity, but also allows for the isolation and testing of large features separately, which can later be merged into an integrated codebase.

The most popular second generation SCM is currently Subversion, which is designed as a drop-in replacement for CVS. It offers several advantages over CVS including tracking commits as a single revision, which avoids file update collisions that can corrupt the CVS repository state.

The third generation of version control is distributed version control systems (DVCS). In DVCS, every developer has a copy of the entire repository along with the full history stored locally. Just like in a second generation version control system, you check out a copy of the repository, make changes, and check it back in. However, to integrate those changes with other developers you sync your entire repository in a peer-to-peer fashion.

There were several early DVCS systems including Arch, Monotone, and Darcs, but DVCS became popularized by Git and Mercurial. Git was developed as a direct response to the Linux team’s need for a stable and reliable version control system that could support the scale and requirements for open source operating

system development, and it has become the de facto standard for both open-source and commercial version control system usage.

Distributed version control systems offer several advantages over server-based version control:

1. *Working entirely offline:* Since you have a local copy of the repository, checking code in and out, merging, and managing branches can all be done without a network connection.
2. *No single point of failure:* Unlike a server-based SCM where only one copy of the entire repository with full history exists, DVCS creates a copy of the repository on every developer's machine, increasing redundancy.
3. *Faster local operations:* Since most version control operations are local to the machine, they are much faster and not affected by network speed or server load.
4. *Decentralized control:* Since syncing the code involves copying the entire repository, this makes it much easier to fork a codebase, and in the case of open source projects can make it much easier to start an independent effort when the main project has stalled or taken an undesirable direction.
5. *Ease of migration:* Converting from most SCM tools into Git is a relatively straightforward operation, and you can retain commit history.

And there are a few disadvantages of distributed version control including:

1. *Slower initial repository sync:* The initial sync includes copying the entire repository history, which can be much slower.
2. *Larger storage requirements:* Since everyone has a full copy of the repository and all history, for very large and/or long running projects this can be a sizable disk requirement.
3. *No ability to lock files:* Server based version control systems offer some support for locking files when a binary file that cannot be merged needs to be edited. With distributed version control systems locking

mechanics cannot be enforced, which means only mergable files like text are suitable for versioning.

Choosing Your Source Control

Hopefully by now you are convinced that using a modern distributed version control system is the way to go. It provides the best capabilities for local and remote development of any size team.

Also, of the commonly used version control systems, Git has become the clear winner in adoption. This is shown clearly by looking at the Google Trends analysis of the most commonly used version control systems as shown in **Figure 2-1**.

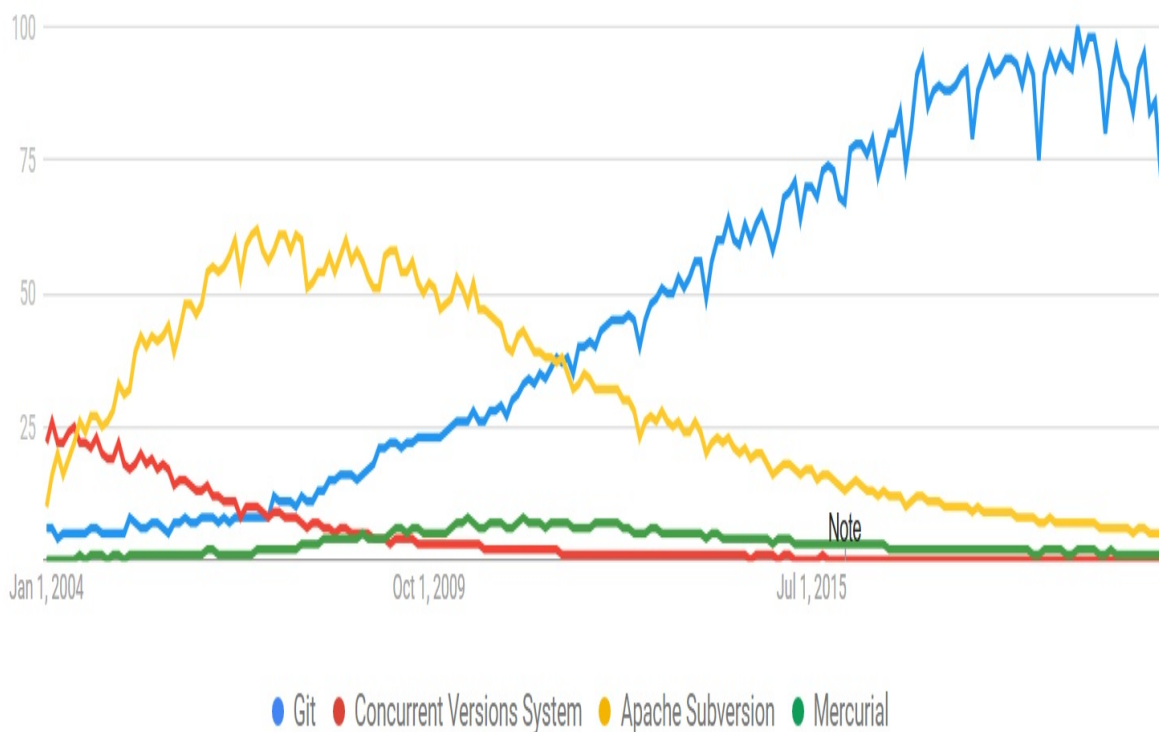


Figure 2-1. Popularity of different version control systems from 2004 through 2020. Source: *Google Trends*

Git has become the de facto standard in the open source community, which means there is a wide base of support for its usage along with a rich ecosystem. However, sometimes it is difficult to convince your boss or peers to adopt new

technologies if they have a deep investment in a legacy source control technology.

Here are some reasons you can use to convince your boss to upgrade to Git:

1. *Reliability*: Git is written like a file system including a proper filesystem check tool (`git fsck`) and checksums to ensure data reliability. And given it is a distributed version control system you probably have your data also pushed to multiple external repositories creating several redundant backups for the data.
2. *Performance*: Git is not the first distributed version control system, but it is extremely performant. It was built from the ground up to support Linux development with extremely large codebases and thousands of developers. Git continues to be actively developed by a large open source community.
3. *Tool support*: There are over 40 different frontends for Git and support in just about every major IDE (IntelliJ, Visual Code, Eclipse, Netbeans, etc.) so you are unlikely to find a development platform that does not fully support it.
4. *Integrations*: Git has first class integrations with IDEs, issue trackers, messaging platform, continuous integration servers, security scanners, code review tools, dependency management, and cloud platforms.
5. *Upgrade Tools*: There are migration tools to ease the transition from other VCS systems to Git, such as `git-svn` that supports bidirectional changes from Subversion to Git, or the Team Foundation Version Control (TFVC) repository import tool for Git.

In summary, there is not much to lose by upgrading to Git, and a lot of additional capabilities and integrations to start to take advantage of.

Getting started with Git is as simple as **downloading a release** for your development machine and creating a local repository. However, the real power comes in collaboration with the rest of your team, and this is most convenient if you have a central repository to push changes to and collaborate with.

There are several companies that offer commercial Git repos that you can self

host or run on their cloud platform. These include Amazon CodeCommit, Assembla, Azure DevOps, GitLab, Phabricator, SourceForge, GitHub, RhodeCode, Bitbucket, Gitcolony, and others. According to data from the 2020 JetBrains Developer Ecosystem Report shown in [Figure 2-2](#), these Git-based source control systems accounted for over 96% of the commercial source control market.

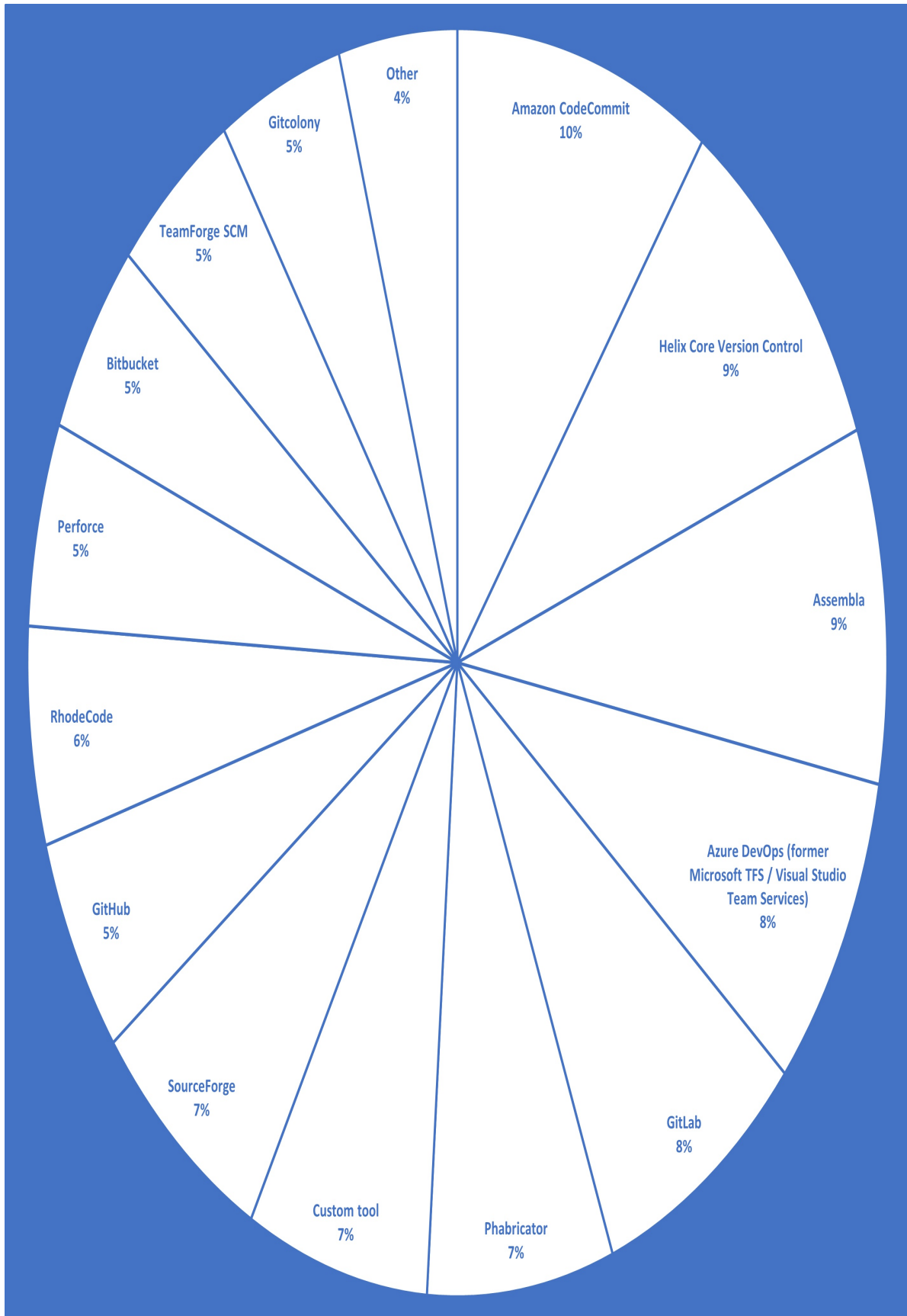


Figure 2-2. Data from the JetBrains State of the Developer Ecosystem 2020 report on usage of Version Control Services *The State of Developer Ecosystem 2020*, JetBrains s.r.o., CC BY 4.0

All of these version control services offer some additional services on top of basic version control, including capabilities like:

- Collaboration
 - *Code reviews*: Having an efficient system for code reviews is important to maintain code integrity, quality, and standards.
 - *Advanced pull request/merge features*: Many vendors implement advanced features on top of Git that help with multirepository and team workflows for more efficient change request management.
 - *Workflow automation*: Approvals in a large organization can be both fluid and complicated, so having automation of team and corporate workflows improves efficiency.
 - *Team comments/discussions*: Effective team interaction and discussions that can be tied to specific pull requests and code changes help to improve communication within and around the team
 - *Online editing*: In-browser IDEs allow for collaboration on source code from anywhere, on almost any device. GitHub even recently released Codespaces (<https://github.com/features/codespaces>) to give you a fully functional development environment hosted by GitHub.
- Compliance/Security
 - *Tracking*: Being able to track the code history is a core feature of any version control system, but often additional compliance checks and reports are required.
 - *Auditing changes*: For control and regulatory purposes, it is often required to audit the changes to a codebase, so having tools to automate this can be helpful.

- *Permissions management*: Fine grained roles and permissions allow for restricting access to sensitive files or codebases.
 - *Bill of materials*: For auditing purposes, a full list of all software modules and dependencies is often required, and can be generated off of the source code.
 - *Security vulnerability scanning*: Many common security vulnerabilities can be uncovered by scanning the codebase and looking for common patterns that are used to exploit deployed applications. Using an automated vulnerability scanner on the source code can help to identify vulnerabilities early in the development process.
- Integration
 - *Issue tracking*: By having tight integration with an issue tracker, you can tie specific changesets to a software defect, making it easier to identify the version a bug is fixed in and trace any regressions.
 - *CI/CD*: Typically a continuous integration server will be used to build the code checked in to source control. A tight integration makes it easier to kick off builds, report back on success and test results, and automate promotion and/or deployment of successful builds.
 - *Binary package repository*: Fetching dependencies from a binary repository and storing build results provides a central place to look for artifacts and to stage deployments.
 - *Messaging integration*: Team collaboration is important to a successful development effort, and making it easy to discuss source files, check-ins, and other source control events simplifies communication with platforms like Slack, MS Teams, Element, etc.
 - *Clients (Desktop/IDE)*: There are a lot of free clients and plug-ins for various IDEs that allow you to access your source

control system including open-source clients from GitHub, BitBucket, and others.

When selecting a version control service it is important to make sure that it fits into the development workflow of your team, integrates with other tools that you already use, and fits into your corporate security policies. Often companies have a version control system that is standardized across the organization, but there may be benefits to adopting a more modern version control system, especially if the corporate standard is not a distributed version control system like Git.

Making Your First Pull Request

To get a feel for how version control works, we are going to run through a simple exercise to create your first pull request to the official book repository on GitHub. We have a section of the readme file dedicated to reader comments, so you can join the rest of the readers in showing your accomplishment in learning modern DevOps best practices!

This exercise doesn't require installing any software or require using the command line, so it should be easy and straightforward to accomplish. Finishing this exercise is highly recommended so you understand the basic concepts of distributed version control that we go into more detail later in the chapter.

To start with you will need to navigate to the book repository at <https://github.com/mjmckay/devops-tools-for-java-developers>. For this exercise you will need to be logged in so you can create a pull request from the web user interface. If you don't already have a GitHub account, it is easy and free to sign up and get started.

The DevOps Tools for Java Developers repository GitHub page is shown in **Figure 2-3**. The GitHub UI shows the root files and the contents of a special file called README.md by default. We are going to make edits to the readme file, which is coded in a visual text language called Markdown.

Since we only have read access to this repository, we are going to create a personal clone of the repository, known as a *fork*, that we can freely edit to make and propose the changes. Once you are logged in to GitHub you can start this process by clicking the *Fork* button highlighted in the upper right corner.

mjmckay/devops-tools-for-java

+

—

□

×

←

→

↺

🔒

https://github.com/mjmckay/devops-tools-for-java

Work

...

🇺🇸

🔄

2

55°

🔍

📺

🔊

🔧

☰

Search or jump to...

/

Pulls

Issues

Marketplace

Explore

+

mjmckay / devops-tools-for-java-developers

👁 Unwatch

3

★ Unstar

1

Fork

0

<> Code

! Issues

🔗 Pull requests

▶ Actions

📁 Projects

📖 Wiki

🛡 Security

...

main

Go to file

Add file

Code

About

steveonjava Update README.md ...

2 minutes ago

🕒 4

build-script-examples

Initial build script examples

14 days ago

README.md

Update README.md

2 minutes ago

README.md

devops-tools-for-java-developers

DevOps Tools for Java Developers resources

DevOps Tools for Java Developers Visitor Log

Add your own visitor comment below. Detailed exercise in Chapter 4.

About

DevOps Tools for Java Developers resources

[Readme](#)

Releases

No releases published

[Create a new release](#)

Packages

No packages published

[Publish your first package](#)

Contributors

2

mjmckay Melissa McKay

steveonjava Stephen C...

Figure 2-3. The GitHub repository for the DevOps Tools for Java Developers book samples

Your new fork will get created under your personal account at GitHub. Once your fork is created, complete the following steps to open the web-based text editor:

1. Click the *README.md* file that you want to edit to see the details page
2. Click the pencil icon on the details page to edit the file

Once you have entered the editor you will see the web-based text editor shown in **Figure 2-4**. Scroll down to the section with the visitor log, and add your own personal comment to the end to let folks know you completed this exercise.

Website page showing the GitHub text file editor

Figure 2-4. The GitHub web-based text editor that you can use to make quick changes to files

The recommended format for visitor log entries is:

- Name (@optional_twitter_handle): Visitor comment

If you want to be fancy on the twitter handle and link to your profile, the markdown syntax for twitter links is as follows:

- [@twitterhandle](<https://twitter.com/twitterhandle>)

To check your changes you can click the “Preview changes” tab, which will show what the rendered output is once inserted into the original readme.

Once you are satisfied with your changes, scroll down to the code commit section shown in **Figure 2-5**. Enter a helpful description for the change to explain what you updated. Then go ahead and click the “Commit changes” button.

For this example we will simply commit to the main branch, which is the default. However, if you were working in a shared repository you would commit your pull request to a feature branch that can be integrated separately.

Website page showing the code commit form




Figure 2-5. Example of how to use the GitHub UI to commit changes to a repository that you have write access to

After you have made a change to your forked repository, you can then submit this as a pull request for the original project. This will notify the project maintainers (in this case, the book authors) that a proposed change is waiting for review and let them choose whether or not to integrate it into the original project.

To do this, go to the “Pull requests” tab in the GitHub user interface. This screen has a button to create a “New pull request” that will present you with a choice of the “base” and “head” repository to be merged, as shown in [Figure 2-6](#).

In this case since you only have one change, the default repositories should be selected correctly. Simply click the “Create pull request” button and a new pull request against the original repository will be submitted for review.

Website page showing the GitHub pull request UI




Figure 2-6. User interface for creating a pull request from a forked repository

This completes your submission of a pull request! Now it is up to the original repository owners to review and comment on, or accept/reject the pull request. While you don't have write access to the original repository to see what this looks like, **Figure 2-7** shows you what will be presented to the repository owners.

Once the repository owners accept your pull request, your custom visitor log greeting will be added to the official book repository.

Website page showing the GitHub merge resolution UI

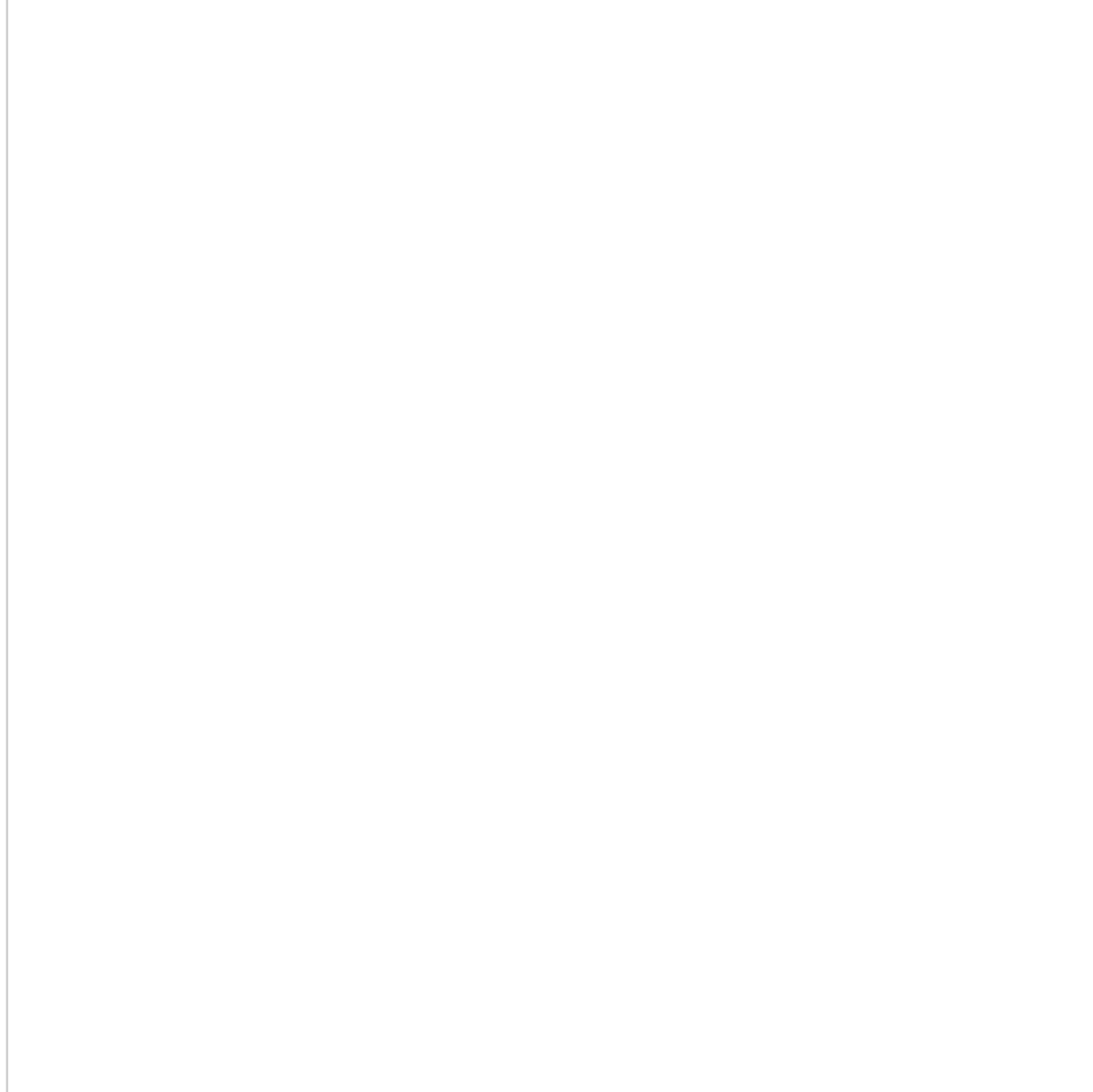


Figure 2-7. The repository owner user interface for merging in the resulting pull request

This workflow is an example of the fork and pull request collaboration model for handling project integration. We will talk a bit more about different collaboration patterns and what sort of projects and team structures they are most suitable for.

Git Tools

In the previous section we showed an entire web-based workflow for Git using the GitHub UI. However, when there are decisions and management

the GitHub UI. However, other than code reviews and repository management, most developers will spend the majority of their time in one of the client-based user interfaces to Git.

The available client interfaces for Git can be broadly split into the following categories:

- *Command line*: There is an official Git command line client that may already be installed on your system, or is easily added.
- *GUI clients*: The official Git distribution comes with a couple open-source tools that can be used to more easily browse your revision history or to structure a commit. Also, there are several third-party free and open-source Git tools that can make working with your repository easier.
- *Git IDE plug-ins*: Often you need to go no farther than your favorite IDE to work with your distributed source control system. Many major IDEs have Git support packaged by default or offer a well supported plug-in.

Git Command Line Basics

The Git command line is the most powerful interface to your source control system, allowing for all local and remote options to manage your repository. You can check to see if you have the Git command line installed by typing the following on the console:

```
git --version
```

If you have Git installed it will return the operating system and version that you are using similar to this:

```
git version 2.26.2.windows.1
```

However, if you don't have Git installed it, here's the easiest way to get it on different platforms:

- Linux distributions:

- *Debian-based*: `sudo apt install git-all`
- *RPM-based*: `sudo dnf install git-all`
- macOS
 - Running `git` on macOS 10.9 or later will ask you to install it.
 - Another easy option is to install **GitHub Desktop**, which installs and configures the command line tools.
- Windows
 - The easiest way is to simply install GitHub Desktop, which installs the command line tools as well.
 - Another option is **Git for Windows**

Regardless of which approach you use to install Git, you will end up with the same great command line tools, which are well supported across all desktop platforms.

To start with, it is helpful to understand the basic Git commands. The diagram in **Figure 2-8** shows a typical repository hierarchy with one central repository and three clients who have cloned it locally. Notice that every client has a full copy of the repository and also a working copy where they can make changes.

Figure 2-8. Diagram of a typical central server pattern for distributed version control collaboration

Some of the Git commands that allow you to move data between repositories and also the working copy are shown. Now let's go through some of the most common commands that are used to manage your repository and collaborate in Git.

Repository management:

- `clone` - This command is used to make a connected copy of another local or remote repository on the local filesystem. For those coming from a concurrent version control system like CVS or Subversion, this command serves a similar purpose to `checkout`, but is semantically different in that it creates a full copy of the remote repository. All of the clients in [Figure 2-8](#) would have cloned the central server to begin.
- `init` - Init creates a new, empty repository. However, most of the time you will start by cloning an existing repository.

Changeset management:

- `add` - This adds file revisions to version control, which can be either a new file or modifications to an existing file. This is different than the `add` command in CVS or Subversion in that it does not *track* the file and needs to be called every time the file changes. Make sure to call `add` on all new and modified files before committing.
- `mv` - Renames or moves a file/directory, while also updating the version control record for the next commit. It is similar in use to the `mv` command in Unix and should be used instead of filesystem commands to keep version control history intact.
- `restore` - This allows you to restore files from the Git index in the case where they are deleted or erroneously modified.
- `rm` - Removes a file or directory, while also updating the version control record for the next commit. It is similar in use to the `rm` command in Unix and should be used instead of filesystem commands to keep version control history intact.

History control:

- **branch** - With no arguments this command lists all of the branches in the local repository. It can also be used to create a new branch or delete branches.
- **commit** - Used to save changes in the working copy to the local repository. Before running commit make sure to register all your file changes by calling **add**, **mv** and **rm** on files that have been added, modified, renamed, or moved. You also need to specify a commit message that can be done on the command line with the **-m** option or if omitted a text editor (such as **vi**) will be spawned to allow you to enter a message.
- **merge** - Merge joins changes from the named commits into the current branch. If the merged in history is already a descendant of the current branch then a “fast-forward” is used to combine the history sequentially. Otherwise a merge is created the combines the history and where there are conflicts the user is prompted to resolve. This command is also used by **git pull** to integrate changes from the remote repository.
- **rebase** - Rebase replays the commits from your current branch on the upstream branch. This is different from **merge** in that the result will be a linear history rather than a merge commit, which can keep the revision history easier to follow. The disadvantage is that rebase creates entirely new commits when it moves the history, so if the current branch contains changes that have previously been pushed you are rewriting history that other clients may depend upon.
- **reset** - This can be used to revert the HEAD to a previous state, and has several practical uses such as reverting an **add** or undoing a commit. However, if those changes have been pushed remotely this can cause problems with the upstream repository. Use with care!
- **switch** - Switches between branches for the working copy. If you have changes in the working copy this can result in a three-way merge, so it is often better to commit or stash your changes first. With the **-c**

command this command will create a branch and immediately switch to it.

- `tag` - Allows you to create a tag on a specific commit that is signed by PGP. This uses the default e-mail address's PGP key. Since tags are cryptographically signed and unique, they should not be reused or changed once pushed. Additional options on this command allow for deleting, verifying, and listing tags.
- `log` - This shows the commit logs in a textual format. It can be used for a quick view of recent changes, and supports advanced options for the history subset shown and formatting of the output. Later in this chapter we also show how to visually browse the history using tools like `gitk`.

Collaboration:

- `fetch` - Fetch pulls the history from a remote repository into the local repository, but makes no attempt to merge it with local commits. This is a safe operation that can be performed at any time and repeatedly without causing merge conflicts or affecting the working copy.
- `pull` - This command is equivalent to a `git fetch` followed by `git merge FETCH_HEAD`. It is very convenient for the common workflow where you want to grab the latest changes from a remote repository and integrate it with your working copy. However, if you have local changes it can cause merge conflicts that you will be forced to resolve. For this reason, it is often safer to `fetch` first and then decide if a simple merge will suffice.
- `push` - This command sends changes to the upstream remote repository from the local repository. Use this after a `commit` to push your changes to the upstream repository so other developers can see your changes.

Now that you have a basic understanding of the Git commands, let's put this knowledge to practice.

Git Command Line Tutorial

To demonstrate how to use these commands, we will go through a simple example to create new local repository from scratch. For this exercise we are assuming you are on a system with a Bash-like command shell. This is the default on most Linux distributions as well as macOS. If you are on Windows you can do this via Windows PowerShell, which has sufficient aliases to emulate Bash for basic commands.

If this is your first time using Git, it is a good idea to put in your name and e-mail, which will be associated with all of your version control operations. You can do this with the following commands:

```
git config --global user.name "Put Your Name Here"  
git config --global user.email "your@email.address"
```

After you have configured your personal information go to a suitable directory to create your working project. First create the project folder and initialize the repository:

```
mkdir tutorial  
cd tutorial  
git init
```

This will create the repository and initialize it so you can start tracking revisions of files. Let's create a new file that we can add to revision control:

```
echo "This is a sample file" > sample.txt
```

To add this file to revision control, use the `git add` command as follows:

```
git add sample.txt
```

And you can add this file to version control by using the `git commit` command as follows:

```
git commit sample.txt -m "First git commit!"
```

Congratulations on making your first command line commit using Git! You can double check to make sure that your file is being tracked in revision control by

using the `git log` command, which should return output similar to the following:

```
commit 0da1bd4423503bba5ebf77db7675c1eb5def3960 (HEAD -> master)
Author: Stephen Chin <steveonjava@gmail.com>
Date:   Sat Oct 31 04:19:08 2020 -0700
```

First git commit!

From this you can see some of the details that Git stores in the repository, including branch information (the default branch being `master`), and revisions by global unique identifiers (GUIDs). Though there is a lot more you can do from the command line, it is often easier to use a Git Client built for your workflow or IDE integration that is designed for a developer workflow. The next couple sections will talk about these client options.

Git Clients

There are several free and open source clients that you can use to work with Git repos more easily and are optimized for different workflows. Most clients do not try to do everything, but specialize in visualizations and functionality for specific workflows.

The default Git installation comes with a couple handy visual tools that make committing and viewing history easier. These tools are written in Tcl-Tk, are cross-platform, and are easily launched from the command line to supplement the Git CLI.

The first tool, Gitk, provides an alternative to the command line for navigating, viewing, and searching the Git history of your local repository. The user interface for Gitk showing the history for the ScalaFX open-source project is shown in [Figure 2-9](#).

Figure 2-9. The bundled Git history viewer application

The top pane of Gitk displays the revision history with branching information drawn visually, which can be very useful for deciphering complicated branch history. Below this are search filters that can be used to find commits containing specific text. Finally, for the selected changeset, you can see the changed files and a textual diff of the changes, which is also searchable.

The other tool that comes bundled with Git is Git Gui. Unlike Gitk, which only shows information about the repository history, Git Gui allows you to modify the repository by executing many of the Git commands including commit, push, branch, merge, and others.

The user interface for Git Gui editing the source code repository for this book is shown in **Figure 2-10**. On the left side all of the changes to the working copy are shown with the unstaged changes on top and the files that will be included in the next commit on the bottom. The details for the selected file are shown in the right side with the full file contents for new files, or a diff for modified files. At the bottom-right buttons are provided for common operations like Rescan, Sign Off, Commit, and Push. Further commands are available in the menu for advanced operations like branching, merging, and remote repository management.

Screenshot of the bundled Git UI for reviewing and committing code

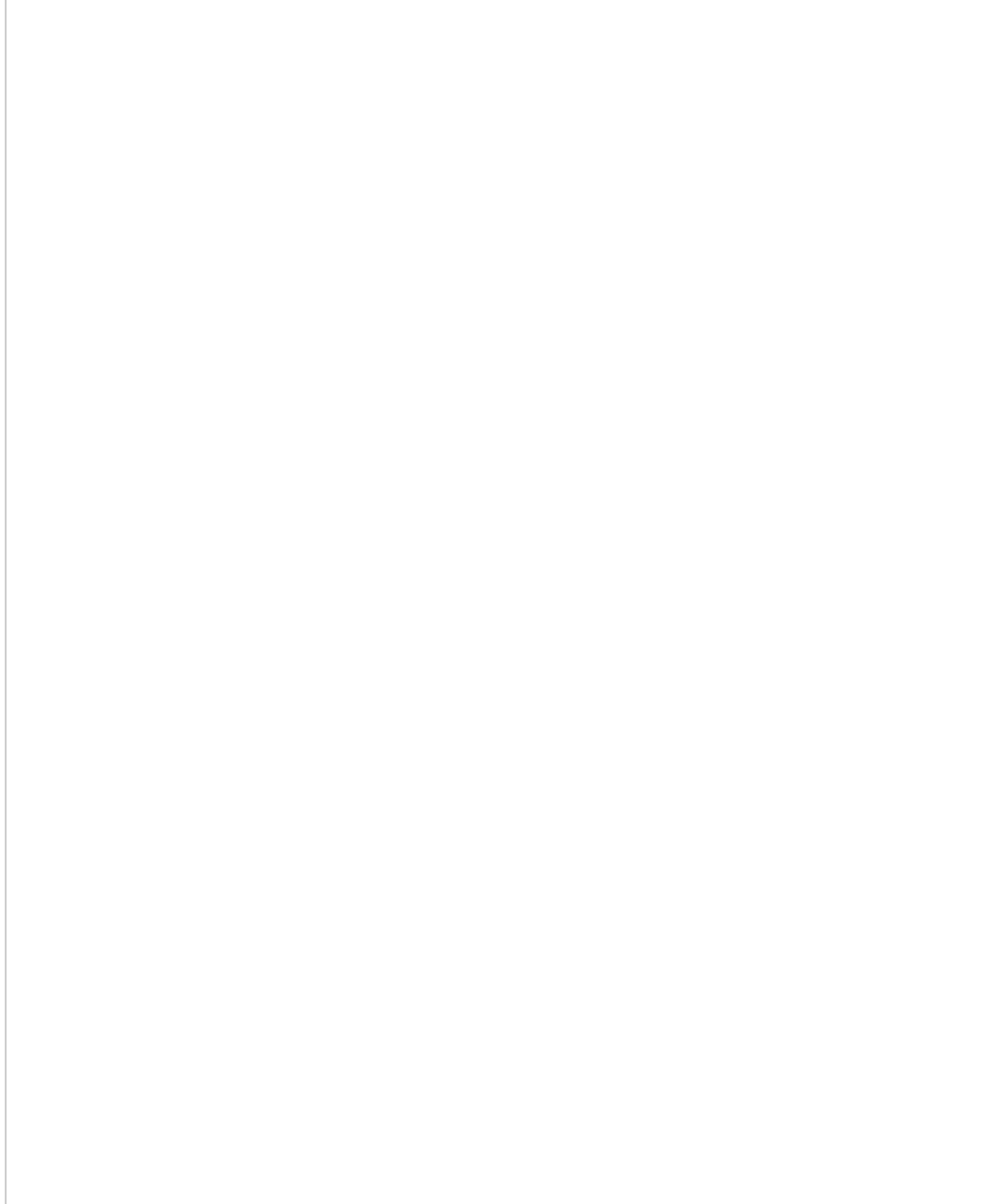


Figure 2-10. The bundled Git collaboration application

Git Gui is an example of a workflow driven user interface for Git. It doesn't expose the full set of functionality available on the command line, but is very convenient for the commonly used Git workflows.

As of this writing, the bundled Git GUI is available on Windows, Mac OS X, and Linux.

Another example of a workflow driven user interface is GitHub Desktop. This is the most popular third-party GitHub user interface, and as mentioned earlier, also conveniently comes bundled with the command line tools so you can use it as an installer for the Git CLI and aforementioned bundled GUIs.

GitHub Desktop is very similar to Git Gui, but is optimized for integration with GitHub's service, and the user interface is designed to make it easy to follow workflows similar to GitHub Flow. The GitHub Desktop user interface editing the source repository for another great book, *The Definitive Guide to Modern Client Development*, is shown in **Figure 2-11**.

Screenshot of the GitHub Desktop user interface

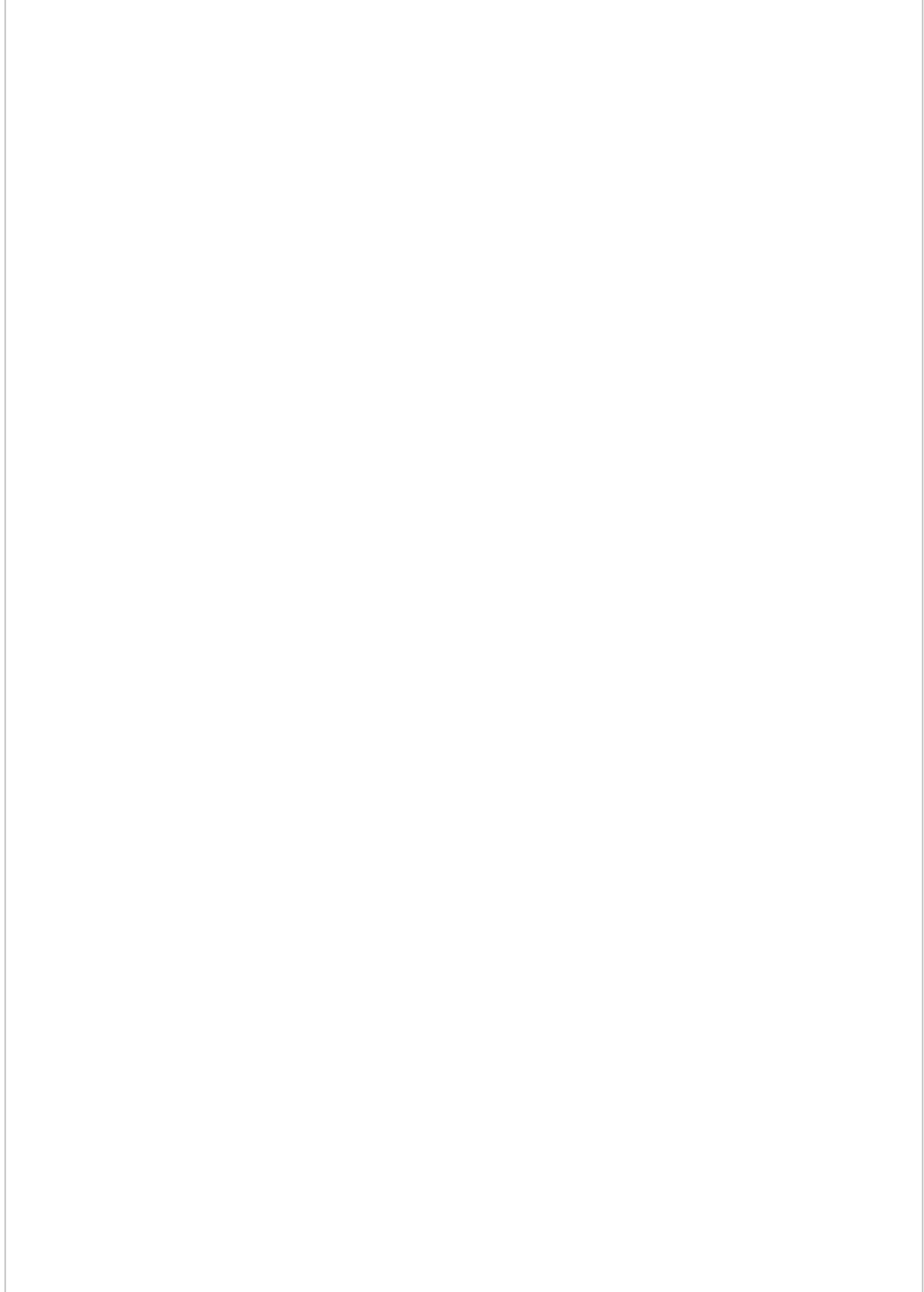


Figure 2-11. GitHub's open source desktop client

In addition to the same sort of capabilities to view changes, commit revisions, and pull/push code as Git Gui, GitHub Desktop has a bunch of advanced features that make managing your code much easier:

- Attributing commits with collaborators
- Syntax highlighted diffs
- Image diff support
- Editor and shell integration
- Checking out and seeing CI status on pull requests

GitHub Desktop can be used with any Git repo, but has features tailored specifically for use with GitHub hosted repositories. Here are some other popular Git tools:

- *SourceTree*: A free, but proprietary, Git client made by Atlassian. It is a good alternative to GitHub Desktop and only has a slight bias towards Atlassian's Git service, BitBucket.
- *GitKraken*: A commercial and featureful Git client. It is free for open source developers, but paid for commercial use.
- *TortoiseGit*: A free, GPL licensed, Git client based on TortoiseSVN. The only downside is that it is Windows only.
- *Others*: A full list of Git GUI clients is maintained on the [Git website](#).

Git desktop clients are a great addition to the arsenal of available source control management tools you have available. However, the most useful Git interface may already be at your fingertips right inside your IDE.

Git IDE Integration

Many integrated development environments (IDEs) include Git support either as a standard feature, or as a well supported plug-in. Chances are that you need to go no further than your favorite IDE to do basic version control operations like adding, moving, and removing files, committing code, and pushing your changes

to an upstream repository.

One of the most popular Java IDEs is JetBrains IntelliJ. It has both a Community Edition that is open source as well as a commercial version with additional features for enterprise developers. The IntelliJ Git support is full featured with the ability to sync changes from a remote repository, track and commit changes performed in the IDE, and integrate upstream changes. The integrated commit tab for a Git changeset is shown in **Figure 2-12**

Screenshot of the IntelliJ commit tab

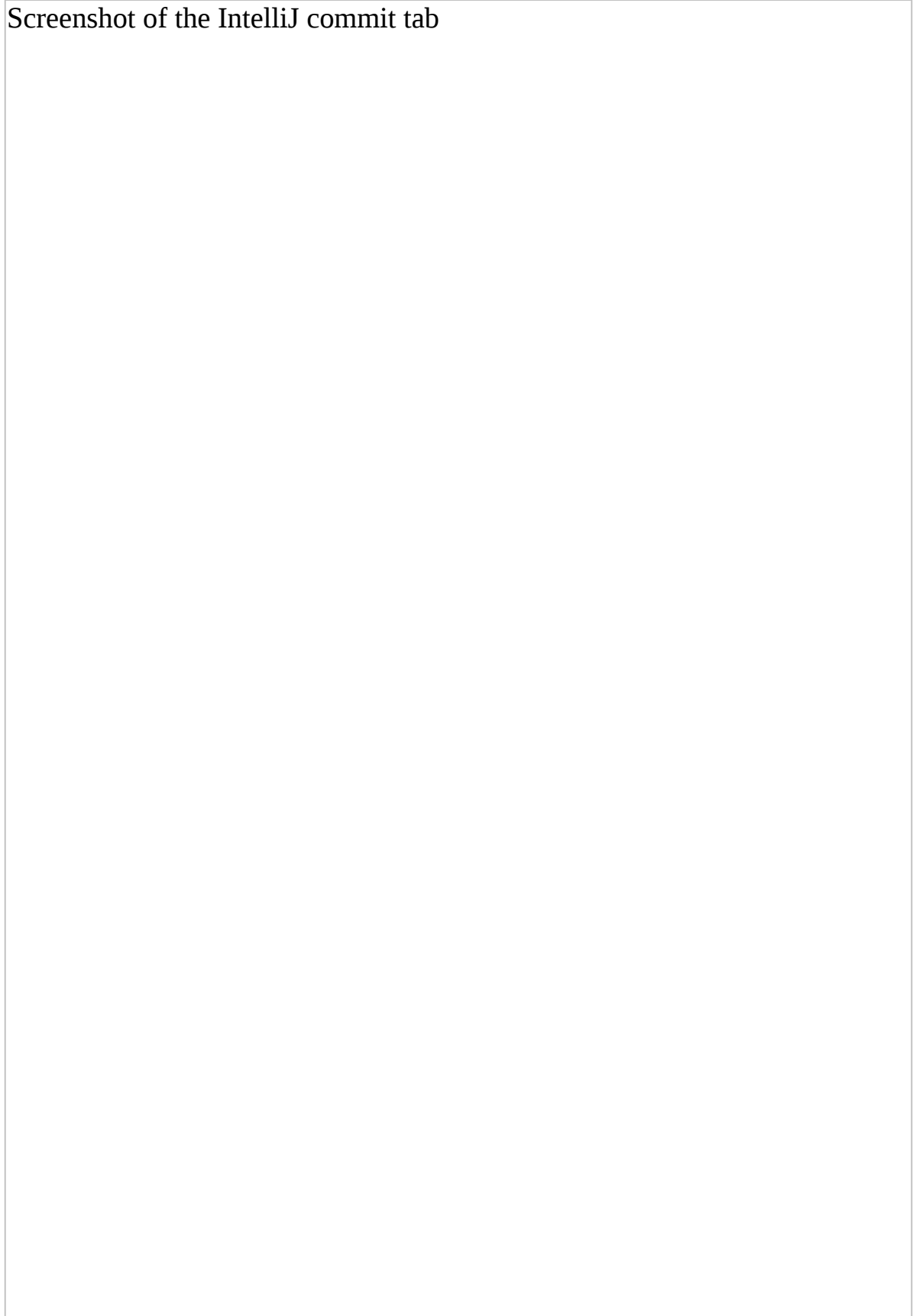


Figure 2-12. IntelliJ Commit tab for managing working copy changes

IntelliJ offers a rich set of features that you can use to customize the Git behavior to your team workflow. For example, if your team prefers a git-flow or GitHub Flow workflow you can choose to merge on update (more details on Git workflows in the next section). However, if your team wants to keep a linear history as prescribed in OneFlow you can choose to rebase on update instead. IntelliJ also supports both native credential provider as well as the open-source Keepass password manager.

Another IDE that offers great Git support is Eclipse, which is a fully open-source IDE that has strong community support and is run by the Eclipse Foundation. The Eclipse Git support is provided by the EGit project, which is based on JGit, a pure Java implementation of the Git version control system.

Due to the tight integration with the embedded Java implementation of Git, Eclipse has the most full-featured Git support. From the Eclipse user interface you can accomplish almost everything that you would normally have to do from the command-line including rebasing, cherry-picking, tagging, patches, and more. The rich set of features is obvious from the settings dialog shown in **Figure 2-13** which has twelve pages of configuration for how the Git integration works and is supported by a user guide, which is almost a book itself at 161 pages in length.

Screenshot of the Eclipse settings dialog for Git configuration

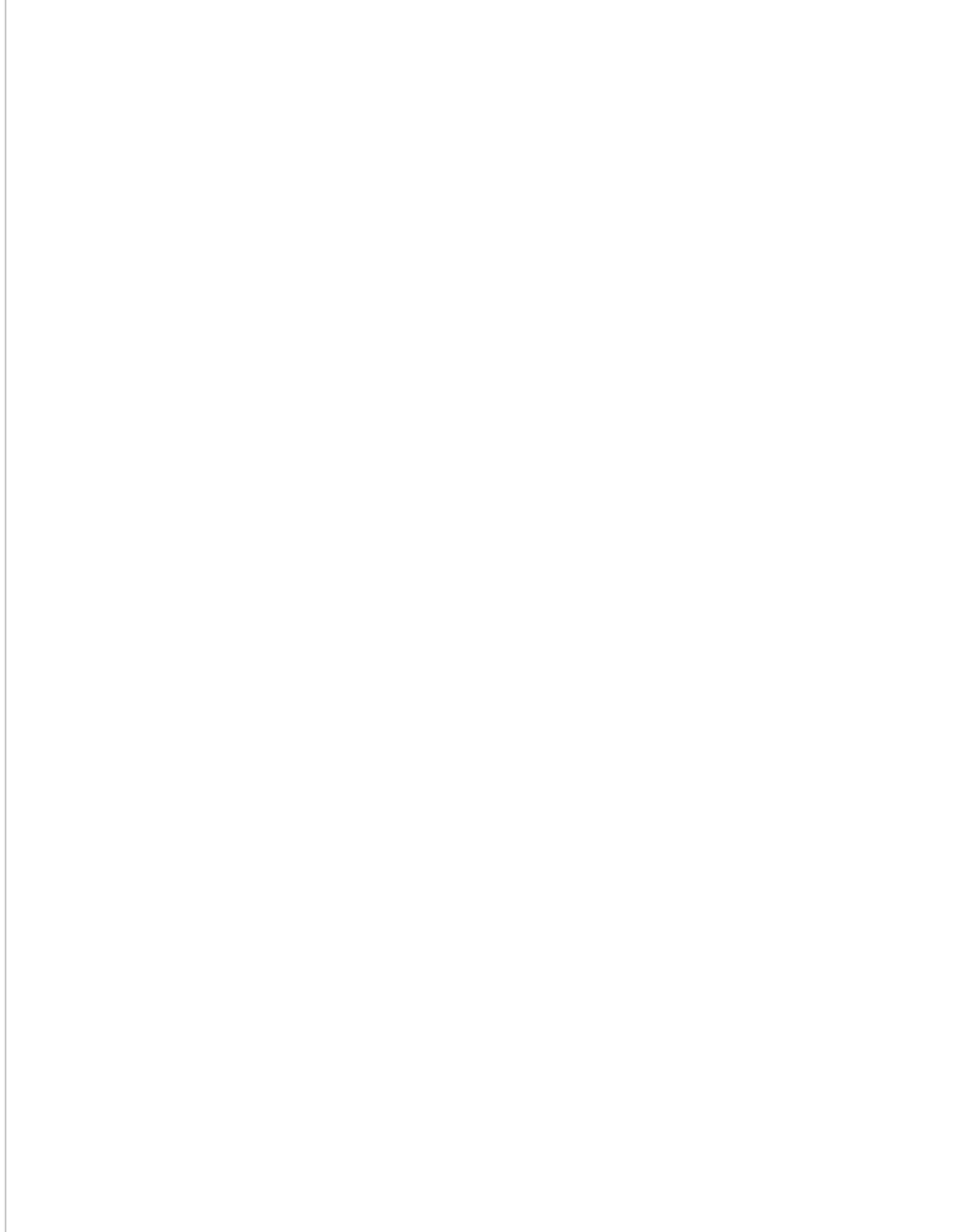


Figure 2-13. Eclipse Settings dialog for Git configuration

Some other Java IDEs that you can expect great Git support from include:

- *NetBeans*: Offers a git plug-in that fully supports workflow from the IDE
- *Visual Studio Code*: Supports Git along with other version control systems out of the box
- *BlueJ*: A popular learning IDE built by King's College London also supports Git in its team workflows
- *JDeveloper*: While it doesn't support complicated workflows, JDeveloper does have basic support for cloning, committing, and pushing to Git repos

So far in this chapter you have added a whole set of new command line, desktop, and integrated tools to your arsenal to work with Git repos. This range of community and industry supported tools means that no matter what your operating system, project workflow, or even team preference is, you will find full tooling support to be successful with your source control management. The next section goes into more detail on different collaboration patterns that are well supported by the full range of Git tools.

Git Collaboration Patterns

Distributed version control systems have a proven track record of scaling to extremely large teams with hundreds of collaborators. At this scale it is necessary to agree upon uniform collaboration patterns that help the team to avoid rework, large and unwieldy merges, and to reduce the amount of time blocked or administering the version control history.

Most projects follow a central repository model where a single repository is designated as the official repository for integrations, builds, and releases. Even though a distributed version control system allows for non-centralized peer-to-peer exchanges of revisions, these are best reserved for short lived efforts between a small number of developers. For any large project having a single system of truth is important and requires one repository that everyone agrees is the official codeline.

For open source projects it is common to have a limited set of developers who have write access to the central repository and other committers are

have write access to the central repository and other contributors are recommended to “fork” the project and issue pull requests to have their changes included. Best practices are to have small pull requests, and to have someone other than the pull request creator accept it. This scales well to projects with thousands of contributors, and allows for review and oversight from a core team when the codebase is not well understood.

However, for most corporate projects a shared repository with a single master branch is preferred. The same workflow with pull requests can be used to keep a central or release branch clean, but this simplifies the contribution process and encourages more frequent integration, which reduces the size and difficulty of merging in changes. For teams on tight deadlines or following an Agile process with short iterations, this also reduces risk of last minute integration failures.

The last best practice that is employed by most teams is to use branches to work on features, which then get integrated back into the main codeline. Git makes it inexpensive to create short lived branches, so it is not uncommon to create and merge back in a branch for work that only takes a couple hours. The risk with creating long-lived feature branches is that if they diverge too much from the main trunk of code development, they can become difficult to integrate back in.

Following these general best practices for distributed version control, there are several collaboration models that have emerged. They share a lot of commonalities, and primarily diverge in their approach to branching, history management, and integration speed.

git-flow

Git-flow is one of the earliest git workflows inspired by a [blog post](#) from Vincent Driessen. It laid the groundwork for later Git collaboration workflows like GitHub Flow; however, git-flow is a more complicated workflow than most projects require and can add additional branch management and integration work.

Key attributes include the following:

- *Development branches*: Branch per feature
- *Merge strategy*: No fast forward merges

- *Rebasing history*: No rebasing
- *Release strategy*: Separate release branch

In git-flow there are two long-lived branches; one for development integration called *develop* and another for final releases called *master*. Developers are expected to do all of their coding in “feature branches” that are named according to the feature they are working on and integrate that with the *develop* branch once complete. When the *develop* branch has the features necessary for a release, a new release branch is created that is used to stabilize the codebase with patches and bugfixes.

Once the release branch has stabilized and is ready for release it is integrated into the *master* branch and given a release tag. Once on the master, only hotfixes are applied, which are small changes managed on a dedicated branch. These hotfixes also need to be applied back to the develop branch and any other concurrent releases that need the same fix. A sample diagram for git-flow is shown in [Figure 2-14](#).

Diagram showing branches and integrations over time

Figure 2-14. Example diagram showing how git-flow manages branches and integration¹

Due to the design decisions on git-flow, it tends to create a very complicated merge history. By not taking advantage of fast-forward merges or rebasing, every integration becomes a commit and the number of concurrent branches can be hard to follow even with visual tools. Also, the complicated rules and branch strategy requires team training and is difficult to enforce with tools, often requiring check-ins and integration to be done from the command line interface.

TIP

Git-flow is best applied to projects that are explicitly versioned and have multiple releases that need to be maintained in parallel. Usually this is not the case for web applications, which only have one *latest* version and can be managed with a single release branch.

If your project sits in the sweet spot where git-flow excels, it is a very well thought out collaboration model, otherwise you may find that a simpler collaboration model will suffice.

GitHub Flow

GitHub Flow is a simplified Git workflow launched in response to the complexity of git-flow by Scott Chacon in another prominent [blog](#). GitHub Flow or a close variant has been adopted by most development teams since it is easier to implement in practice, handles the common case for continuous released web development, and is well supported by tools.

Key attributes include the following:

- *Development branches*: Branch per feature
- *Merge strategy*: No fast forward merges
- *Rebasing history*: No rebasing
- *Release strategy*: No separate release branches

GitHub Flow takes a very simple approach to branch management, using *master* as the main codeline and also the release branch. Developers do all of their work

on short-lived feature branches and integrate them back into master as soon as their code passes tests and code reviews.

TIP

In general, GitHub Flow makes good use of available tooling by having a straightforward workflow with a simple branching strategy and no use of complicated arguments to enable fast-forward merges or replace merges with rebasing. This makes GitHub Flow easy to adopt on any team and use by developers who are not familiar with the team process or are not as familiar with the command line Git interface.

The GitHub Flow collaboration model works well for server-side and cloud deployed applications where the only meaningful version is the latest release. In fact, GitHub Flow recommends that teams continuously deploy to production to avoid feature stacking where a single release build has multiple features that increase complexity and make it harder to determine the breaking change. However, for more complicated workflows with multiple concurrent releases GitHub Flow needs to be modified to accommodate.

Gitlab Flow

Gitlab Flow is basically an extension of GitHub Flow documented on Gitlab's [website](#). It takes the same core design principles about using master as a single long-lived branch and doing the majority of development on feature branches. However, it adds a few extensions to support release branches and history clean-up that many teams have adopted as best practices.

Key attributes include the following:

- *Development branches*: Branch per feature
- *Merge strategy*: Open ended
- *Rebasing history*: Optional
- *Release strategy*: Separate release branches

The key difference between GitHub Flow and Gitlab Flow is the addition of release branches. This is recognition that most teams are not practicing continuous deployment at the level GitHub does. By having release branches this

continuous deployment at the level GitHub does. By having release branches this allows stabilization of code before it gets pushed into production; however, Gitlab Flow recommends making patches to master and then cherry picking them for release rather than having an extra hotfix branch like git-flow.

The other significant different is the willingness to edit history using rebase and squash. By cleaning up the history before committing to master, it is easier to retroactively go back and read the history to discover when key changes or bugs were introduced. However, this involves rewriting the local history and can be dangerous when that history has already been pushed to the central repository.

TIP

Gitlab Flow is a modern take on the Gitlab Flow philosophy to collaboration workflow, but ultimately your team has to decide on the features and branch strategy based on your project's needs.

OneFlow

OneFlow is another collaboration workflow based on git-flow proposed by Adam Ruka and consolidated in a detailed [blog](#). OneFlow makes the same adaptation as GitHub/Gitlab flow in squashing the separate *develop* branch in favor of feature branches and direct integration on main. However, it keeps the release and hotfix branches that are used in git-flow.

Key attributes include the following:

- *Development branches*: Branch per feature
- *Merge strategy*: No fast forward merges without rebase
- *Rebasing history*: Rebasing recommended
- *Release strategy*: Separate release branches

The other big deviation in OneFlow is that it heavily favors modifying history to keep the Git revision history readable. It offers three different merge strategies that have varying levels of revision cleanliness and rollback friendliness:

- *Option #1: Rebase*: This makes the merge history mostly linear and

easy to follow. It has the usual caveat that changesets pushed to the central server should not be rebased and makes it more difficult to rollback changes since they are not captured in a single commit.

- *Option #2: merge -no-ff*: This is the same strategy used in git-flow and has the disadvantage that the merge history is largely non-sequential and difficult to follow.
- *Option #3: rebase + merge -no-ff*: This is a rebase workaround that tacks on an extra merge integration at the end so it can be rolled back as a unit even though it is still mostly sequential.

TIP

OneFlow is a thoughtful approach to a Git collaboration workflow that is created from the experience of developers on large enterprise projects. It can be seen as a modern variant on git-flow that should serve the needs of projects of any size.

Trunk Based Development

All of the aforementioned approaches are variants of the feature branch development model where all active development is done on branches that get merged in to either master or a dedicated development branch. They take advantage of the great support Git has for branch management, but if features are not granular enough, they suffer from the typical integration problems that have plagued teams for decades. The longer the feature branch is in active development, the higher likelihood there is for merge conflicts with other features and maintenance going on in the master branch (or trunk).

Trunk-based development solves this problem by recommending that all development happen on the main branch with very short integrations that occur anytime that tests are passing, but not necessarily waiting for a full feature to be completed.

Key attributes include the following:

- *Development branches*: Optional, but no long-lived branches
- *Merge strategy*: Only if using development branches

- *Rebasing history*: Rebasing recommended
- *Release strategy*: Separate release branches

Paul Hammant is a strong advocate for trunk-based development having setup a [full website](#) and written a book on the topic. While this is not a new approach to collaboration on source control management systems, it is a proven approach to agile development in large teams and works equally well on classic central SCMs like CVS and Subversion to modern distributed version control systems like Git.

Summary

Good source control systems and practices lay the foundation for a solid DevOps approach to building, releasing, and deploying code quickly. In this chapter we discussed the history of source control systems and explained why the world has moved to embrace distributed version control.

This consolidation has built a rich ecosystem of source control servers, developer tools, and commercial integrations. Finally, through the adoption of distributed version control by DevOps thought leaders, best practices and collaboration workflows have been established that you can follow to help make your team successful with adopting as modern SCM.

In the next few chapters we will drill into systems that connect to your source control management system including continuous integration, package management, and security scanning that allow you to rapidly deploy to traditional or cloud-native environments. You are on the way to building a comprehensive DevOps platform that will support whatever workflow you need to meet your quality and deployment objectives.

Chapter 3. Dissecting the Monolith

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors’ raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book. If there is a GitHub repo associated with the book, it will be made active after final publication.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at ccollins@oreilly.com.

The ultimate goal should be to improve the quality of human life through digital innovation.

—Pony Ma Huateng

Through history, humans have been obsessed with analysing, that is, decomposing ideas and concepts into simple or composite parts. It is by combining analysis and synthesis that we can achieve a higher level of understanding.

Aristotle called **Analytics** to “*the resolution of every compound into those things out of which the synthesis is made. For analysis is the converse of synthesis. Synthesis is the road from the principles to those things that derive from the principles, and analysis is the return from the end to the principles.*”¹

Software development follows a similar approach to philosophy; analyse a system into its composite parts, identifying inputs, desired outputs and detail functions. During this process we have come to a realisation that non-business specific functionality is always required to process inputs and to communicate or persist outputs. Making painfully obvious that we could benefit for reusable,

well defined, context bound, atomic functionality that can be shared, consumed, or interconnect to simplify building software.

Allowing developers to focus primarily on implementing business logic to fulfil purposes, like “*meet specific needs of a specific client/business; meet a perceived need of some set of potential users, or for personal use (to automate mundane tasks)*” has been a long held desire.² Forgoing wasting time every day reinventing one of the most reinvented wheels, reliable boilerplate code.

The Microservices pattern has gained a lot of notoriety and momentum in recent years. The reason is simply the promised benefits are outstanding. Avoiding known anti-patterns, adopting best practices, understanding core concepts and definitions is paramount in achieving the benefits of this architectural pattern while reducing the drawbacks of adopting said pattern.

In this chapter we will cover core concepts, known anti-patterns and code examples of microservices written with popular microservice frameworks such as Spring Boot, Micronaut, Quarkus, Helidon.

Monolithic architecture

Traditional what we have now know as Monolithic architecture is the one that delivers or deploys single units or systems, addressing all requirements from a single source application.

In a Monolith, we can identify two concepts, the “*monolith application*” or the “*monolithic architecture*”.

A “*monolith application*” has **only one** deployed instance, responsible to perform any and all steps needed for a specific function. A unique interface point of execution with the client or user.

The “*monolithic architecture*” refers to an application where all requirements are addressed from a **single source** and all parts are delivered as one unit. Components in the monolith, may be interconnected or interdependent rather than loosely coupled. Components may have been designed to restrict the interaction with external clients to explicitly limit the access of *private* functionality.

Granularity and functional specification

Granularity is the aggregation level exposed by a component to other external cooperating or collaborating parts of software. The level of granularity in software depends on several factors, such as the level of confidentiality that must be maintained within a series of components and not be exposed or available to other consumers.

Modern software architectures are increasingly focused on delivering functionality by bundling or combining software components from different sources, resulting or emphasising a fine coarse granularity level of detail. The functionality exposed then to different components, customers, or consumers is greater than in a Monolithic application.

To qualify how independent or interchangeable a module is we should look closely to the following characteristic:

- the number of dependencies.
- the strength of these dependencies.
- and the stability of the modules it depends on.

Any high score number assigned to the previous characteristics should trigger a second review on the modeling and definition of the module.

Cloud Computing

What is cloud computing? According to NIST,

“Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.”³

What is the state of Cloud Computing (*market wise*)?

Cloud infrastructure services spending increased 32% to US\$39.9 billion in the last quarter of 2020, following heightened customer investment with the major cloud service providers and the technology channel. Total expenditure was over

US\$3 billion higher than the last quarter and nearly US\$10 billion more than Q4 2019 according to Canalys data.⁴

Who are the winners?

“Amazon Web Services (AWS) was the leading cloud service provider again in Q4 2020, accounting for 31% of total spend. Microsoft’s Azure growth rate accelerated once again, up by 50% to boost its market share in the global cloud services market to 20% Google Cloud was the third largest cloud service provider in Q4 2020, with a 7% share.”⁵

Is cost effective to migrate as soon as possible? What about usage patterns in relation to pricing?

Studies of reported usage of cloud resources in datacenters show a substantial gap between the resources that cloud customers allocate and pay for (leasing VMs), and actual resource utilization (CPU, memory, and so on), perhaps customers are just leaving their VMs on, but not actually using them.⁶

Service Models

Clouds are separated into service categories used for different types of computing.

Software as a Service (SaaS)

The capability provided to the consumer is to use the provider’s applications running on a cloud infrastructure. The applications are accessible from various client devices through either a thin client interface, such as a web browser, or a program interface. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, storage, or even individual application capabilities, with the possible exception of limited user- specific application configuration settings.

Platform as a Service (PaaS)

The capability provided to the consumer is to deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages, libraries, services, and tools supported by the provider. The consumer does not manage or control the underlying cloud

provision the consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, or storage, but has control over the deployed applications and possibly configuration settings for the application-hosting environment.

Infrastructure as a Service (IaaS)

The capability provided to the consumer is to provision processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software, which can *include operating systems and applications*. The consumer does not manage or control the underlying cloud infrastructure but has control over operating systems, storage, and deployed applications; and possibly limited control of select networking components.

Microservices

Definition

The term ‘Micro Web Services’ was first introduced by Peter Rodgers back in 2005 during a Web Services Edge Conference. Rodgers was making a point for REST services and championing the idea of ‘Software as Micro-Web-Services’.⁷

Microservice architecture – an evolution of Service-oriented Architectures - arranges an application as a collection of loosely coupled and relatively lightweight modular services.

Technically, Microservices is a specialisation of an implementation approach for SOA - Service Oriented Architectures

Loosely coupled system is one in which each of its components has, or makes use of, little or no knowledge of the definitions of other separate components.

Microservices are small components that in contrast to monoliths, allows them to be deployed, scaled, tested independently and has a single responsibility, bounded by context, autonomous and decentralized; Usually built around business capabilities easy to understand, and may be developed using different technology stacks.

How “*small*” should a “*microservice*” be? They should be “*micro*” enough to allow delivering small, self-contained and rigidly enforced atoms of functionality that can coexist, evolve or replace the previous ones according to the business needs.

Each component or service has little or no knowledge of the definitions of other separate components, and all interaction with a service is via its API, which encapsulates its implementation details.

The messaging between these “microservices” uses simple protocols and usually is not data intensive.

Anti-Patterns

The microservice pattern results in significant complexity and is not ideal in all situations. Not only is the system made up of many parts that work independently, but by its very nature, it’s harder to predict how it will perform in the real world.

This increased complexity is mainly due to the potentially thousands of microservices running asynchronously in the distributed computer network. Keep in mind that programs that are difficult to understand are also difficult to write, modify, test and measure. All these concerns will increase the time which teams need to spend understanding, discussing, tracking, and testing interfaces and message formats.

There are several books, articles and papers on this particular topic. I would like to recommend you to visit “*microservices.io*” ⁸ or Mark Richards’ book on “*Microservices AntiPatterns and Pitfalls*” ⁹ and finally but not less important “*On the Definition of Microservice Bad Smells*” ¹⁰

Some of the most common antipatterns are:

API versioning (“*Static Contract Pitfall*”)

APIs need to be semantically versioned to allow services to know whether they are communicating with the right version of the service or whether they need to adapt their communication to a new contract.

Inappropriate service *privacy* interdependency

The microservice requires private data from other services instead of dealing with its own data. A problem that usually is related to a modeling the data issue. One solution that could be consider is merging the microservices.

Multi-purpose *megaservice*

Several business functions implemented in the same service.

Logging

Errors and microservices information are hidden inside each microservice container. The adoption of a distributed logging system should be a priority as find issues in all stages of the software life cycle.

Complex interservice or circular dependencies

A circular service relationship is defined as a relationship between two or more services that are interdependent. Circular dependencies can harm the ability of services to scale or deploy independently, as well as violate the “*Acyclic Dependency Principle (ADP)*”.

***(Missing)* API Gateway**

When microservices communicate directly with each other, or the service consumers also communicate directly with each microservice there is an increase in the complexity and decrease on maintenance of the system. The best practice in this case is to use a *API Gateway*.

An API gateway receives all API calls from clients and then directs them to the appropriate microservice by request routing, composition, and protocol translation. It usually handles the request by calling multiple microservices and aggregating the results to determine the best route. It is also able to translate between web protocols and web-friendly protocols for internal use.

A site can use an API gateway to provide a single endpoint for mobile customers to query all product data with a single request. Consolidates various services such as product information and reviews and combines the results.

The API gateway is the gatekeeper for applications to access data, business logic, or functions, (*RESTful APIs or WebSocket APIs*) that allow real-time two-way communication applications. The API gateway typically handles all the

tasks involved in accepting and processing up to hundreds of thousands of concurrent API calls, including traffic management, CORS support, authorization and access control, choking, management, and API version control.

Sharing too much! (*"I Was Taught to Share"*)

There is a thin line between sharing enough functionality to not repeat ourselves and creating a tangled mess of dependencies that prevents service changes from being separated. If an over-shared service needs to be changed, evaluating proposed changes in the interfaces will eventually lead to a organizational task involving more development teams.

At some point, the choice of redundancy or library extraction into a new, shared service that related microservices can install and develop independently of each other needs to be analyzed.

DevOps & Microservices

Microservices fits perfectly into the DevOps ideals of utilizing small teams to create functional changes to the enterprise's services one step at a time, the idea of breaking large problems into smaller pieces and tackling them systematically. In order to reduce the friction between development, testing and deployment of smaller independent service, a series of continuous delivery pipelines to maintain a steady flow of these stages has to be present.

DevOps is a key factor in the success of this architectural style, providing the necessary organizational changes to minimize coordination between teams responsible for each component and to remove barriers to effective, reciprocal interaction between development and operations teams.

I would strongly dissuade any team that is willing to adopt the microservices pattern to do so without a robust CI/CD infrastructure in place or without a widespread understanding of the basic concepts of pipelines.

Microservice Frameworks

The JVM ecosystem is vast and provides plenty of alternatives for a particular use case. There are dozens of options when it comes to microservices

frameworks and libraries to the point that it can be tricky to pick a winner among candidates. This being said there is a set of candidate frameworks that have gained popularity for several reasons: developer experience, time to market, extensibility, resource (CPU & memory) consumption, startup speed, failure recovery, documentation, 3rd party integrations, and more. These frameworks are in order of appearance: Spring Boot, Micronaut, Quarkus, and Helidon. We'll cover each one of these options in the following sections. Take into account that some of the instructions may require additional tweaks based on newer versions as some of these technologies evolve quite rapidly, for which I strongly advise you to review the documentation of each framework.

Additionally, these examples require Java 11 as a minimum and in case that you'd like to try out Native Image generation you'd also require a installation of GraalVM. There are many ways to get these versions installed in your environment, I recommend using **Sdkman!** to install and manage them. For brevity we'll concentrate on production code alone as one can fill a whole book on a single framework! It goes without saying that you should take care of tests as well. Our goal for each example is to build a trivial Hello World REST service that can take an optional name parameter and reply with a greeting.

If you have not heard of GraalVM before let's just say it's an umbrella project for a handful of technologies that enable the following features:

- A “Just In Time” compiler (JIT for short) written in Java. A JIT compiles code on the fly, transforming interpreted code into executable code. The Java platform has had a handful of JITs , most of them written using a combination of C & C++, Graal happens to be the most modern one written in Java.
- A virtual machine named SubstrateVM that's capable of running hosted languages such as Python, JavaScript, R, on top of the JVM in such a way that the hosted language benefits from tighter integration with JVM capabilities and features.
- Native Image, an utility that relies on “Ahead of Time” compilation (AOT for short) which transforms bytecode into machine executable code. The resulting transformation produces a platform specific binary executable.

All of the four candidate frameworks we'll cover next provide support for GraalVM in one way or another, chiefly relying on GraalVM Native Image to produce platform specific binaries with the goal of reducing deployment size and memory consumption. Be aware that there's a tradeoff between using the Java mode and the GraalVM Native Image mode, as the later can produce binaries with smaller memory footprint and faster startup time but requires longer compilation time; long running Java code will eventually become more optimized (that's one of the key features of the JVM) whereas native binaries cannot be optimized while running; development experience also varies as you may need to use additional tools for debugging, monitoring, measuring, and so forth.

Spring Boot

Spring Boot is perhaps the most well known among the four candidates, as it builds on top of the legacy laid out by the Spring Framework. If developer surveys are to be taken on face value then more than 60% of Java developers have some sort of experience interacting with Spring related projects, making Spring Boot the most popular choice among others.

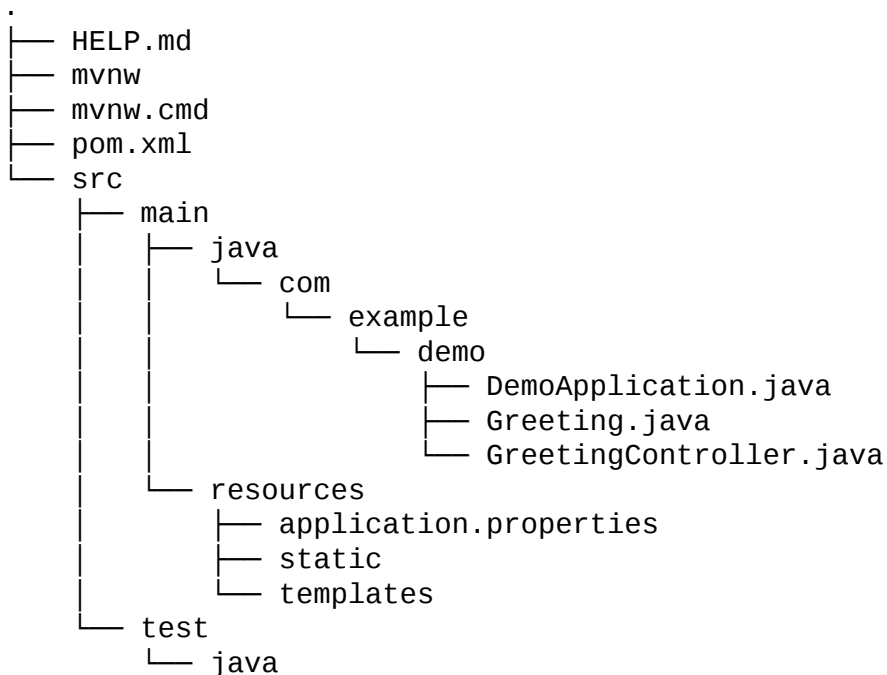
The Spring way lets you assemble applications (or microservices in our case) by composing existing components, customizing their configuration, promising low-cost code ownership as your custom logic is supposedly smaller in size than what the framework brings to the table, and for most organizations that's true. The trick is to search for an existing component that can be tweaked and configured before writing your own. The Spring Boot team makes a point of adding as many useful integrations as needed, from database drivers to monitoring services, logging, journaling, batch processing, report generation, and more.

The typical way to bootstrap a Spring Boot project is by browsing to <https://start.spring.io>, selecting the features you require in your application, then clicking on the "Generate" button. This action creates a Zip file that you can download into your local environment to get started. As you can appreciate on Figure 1. I've selected the Web and Spring Native features. The first feature adds components that let you expose data via REST APIs, the second feature enhances the build with an extra packaging mechanism that can create Native

Images with Graal.

spring-boot

Unpacking the zip file and running the `./mvnw verify` command at the root directory of the project ensures that we have a sound starting point. You'll notice the command will download a set of dependencies if you have not built a Spring Boot application before on your target environment. This is normal Maven behavior. These dependencies won't be downloaded again the next time you invoke a Maven command again, that is, unless dependency versions are updated in the *pom.xml* file. The project structure should look like the following one



We require two additional sources that were not created by the Spring Initializr website: *Greeting.java* and *GreetingController.java*. These two files can be created using your text editor or IDE of choice. The first file *Greeting.java* defines a data object that will be used to render content as JSON as that's a typical format used to expose data via a REST endpoint; other formats are also supported it just happens that JSON support comes out of the box without any additional dependencies required. This file should look like this

```
package com.example.demo;

public class Greeting {
    private final String content;
```

```

    public Greeting(String content) {
        this.content = content;
    }

    public String getContent() {
        return content;
    }
}

```

There's nothing special about this data holder except that it's immutable; depending on your use case you might want to switch to a mutable implementation but for now this will suffice. Next is the REST endpoint itself, defined as a GET call on a `/greeting` path. Spring Boot prefers the *controller* stereotype for this kind of component, no doubt harkening back to the days where Spring MVC (yes, that's Model View Controller) was the preferred option to create web applications. Feel free to use a different filename, but the component annotation must remain untouched

```

package com.example.demo;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class GreetingController {
    private static final String template = "Hello, %s!";

    @GetMapping("/greeting")
    public Greeting greeting(@RequestParam(value = "name",
        defaultValue = "World") String name) {
        return new Greeting(String.format(template, name));
    }
}

```

The controller may take a `name` parameter as input and will use the value `World` where such parameter is not supplied. Notice that the return type of the mapped method is a plain Java type, matter of fact it's the data type we just defined in the previous step. Spring Boot will automatically marshall data from and to JSON based on the annotations applied to the controller and its methods, as well as sensible defaults put in place. If we leave the code as is then the return value of the `greeting()` method will be automatically transformed into a

JSON payload. This is the power of Spring Boot's developer experience, relying on defaults and predefined configuration that may be tweaked as needed.

Running the application can be done by either invoking the `./mvnw spring-boot:run` command which runs the application as part of the build process, or by generating the application JAR and running it manually, that is `./mvnw package` followed by `java -jar target/demo-`

`0.0.1.SNAPSHOT.jar`. Either way an embedded web server will be started listening on port 8080; the `/greeting` path will be mapped to an instance of the *GreetingController*. All that it's left now is to issue a couple of queries, such as

```
// using the default name parameter
$ curl http://localhost:8080/greeting
{"content":"Hello, World!"}

// using an explicit value for the name parameter
$ curl http://localhost:8080/greeting?name=Microservices
{"content":"Hello, Microservices!"}
```

Take note of the output generated by the application while running, on my local environment shows (on average) that the JVM takes 1.6s to startup while the application takes 600ms to initialize. The size of the generated JAR is roughly 17 MB. You may also want to take notes on the CPU & memory consumption of this trivial application. For some time now the use of GraalVM Native Image has been suggested to reduce the startup time and the binary size. Let's see how we can make that happen with Spring Boot.

Remember that we selected the Spring Native feature when the project was created, right? Unfortunately by version 2.5.0 the generated project does not include all required instructions in the *pom.xml* file, we must make a few tweaks. To begin with the JAR created by `spring-boot-maven-plugin` requires a classifier or else the resulting Native Image may not be properly created due to the fact that the application JAR already contains all dependencies inside a Spring Boot specific path that's not handled by the `native-image-maven-plugin` plugin that we also have to configure. The update *pom.xml* file should look like this

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```

<!-- xsi:schemaLocation="http://www.w3.org/2001/XMLSchema-instance
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.5.0</version>
</parent>
<groupId>com.example</groupId>
<artifactId>demo</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>demo</name>
<description>Demo project for Spring Boot</description>
<properties>
  <java.version>11</java.version>
  <spring-native.version>0.10.0-SNAPSHOT</spring-native.version>
</properties>
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.experimental</groupId>
    <artifactId>spring-native</artifactId>
    <version>${spring-native.version}</version>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <configuration>
        <classifier>exec</classifier>
      </configuration>
    </plugin>
    <plugin>
      <groupId>org.springframework.experimental</groupId>
      <artifactId>spring-aot-maven-plugin</artifactId>
      <version>${spring-native.version}</version>
      <executions>
        <execution>
          <id>test-generate</id>

```

```

        <id>test-generate</id>
        <goals>
            <goal>test-generate</goal>
        </goals>
    </execution>
    <execution>
        <id>generate</id>
        <goals>
            <goal>generate</goal>
        </goals>
    </execution>
</executions>
</plugin>
</plugins>
</build>
<repositories>
    <repository>
        <id>spring-release</id>
        <name>Spring release</name>
        <url>https://repo.spring.io/release</url>
    </repository>
</repositories>
<pluginRepositories>
    <pluginRepository>
        <id>spring-release</id>
        <name>Spring release</name>
        <url>https://repo.spring.io/release</url>
    </pluginRepository>
</pluginRepositories>

<profiles>
    <profile>
        <id>native-image</id>
        <build>
            <plugins>
                <plugin>
                    <groupId>org.graalvm.nativeimage</groupId>
                    <artifactId>native-image-maven-
plugin</artifactId>
                    <version>21.1.0</version>
                    <configuration>

<mainClass>com.example.demo.DemoApplication</mainClass>
                </configuration>
            <executions>
                <execution>
                    <goals>
                        <goal>native-image</goal>
                    </goals>
                    <phase>package</phase>
                </execution>
            </executions>

```

```

        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
</profile>
</profiles>
</project>

```

One more step to go before we can give it a try: make sure to have a version of GraalVM installed as your current JDK. The selected version should closely match the version of `native-image-maven-plugin` found in the *pom.xml* file. you also require the `native-image` executable to be installed in your system; this can be done by invoking `gu install native-image`. `gu` is a command provided by the GraalVM installation.

Alright, we're good to go, with all settings in place we can generate a native executable by invoking `./mvnw -Pnative-image package`. You'll notice a flurry of text going through the screen as new dependencies may be downloaded and perhaps a few warnings related to missing classes, that's normal. You'll also notice that the build takes longer than usual and here lies the trade-off of this packaging solution: we are exchanging development time (increasing it) to speed up execution time at production. Once the command finishes you'll notice a new file *com.example.demo.demoapplication* inside the *target* directory. This is the native executable. Go ahead and run it.

Did you notice how fast the startup was? On my environment I get on average a startup time of 0.06 seconds while the application takes 30ms to initialize itself. You might recall these numbers were 1.6s and 600ms when running in the Java mode. This is a serious speed boost! Now have a look at the size of the executable, in my case it's around 78 MB. Oh well looks like some things have grown for the worse, or have they? This executable is a single binary that provides everything that's needed to run the application while the JAR we used earlier requires a Java Runtime to run. The size of a Java Runtime is typically in the 200M range and is composed of multiple files and directories. Of course smaller Java Runtimes may be created with `jlink` in which case that adds up another step during the build process. There's no free lunch.

We stop with Spring Boot for now but know that there's a whole lot more to it that what has been shown here. On to the next framework.

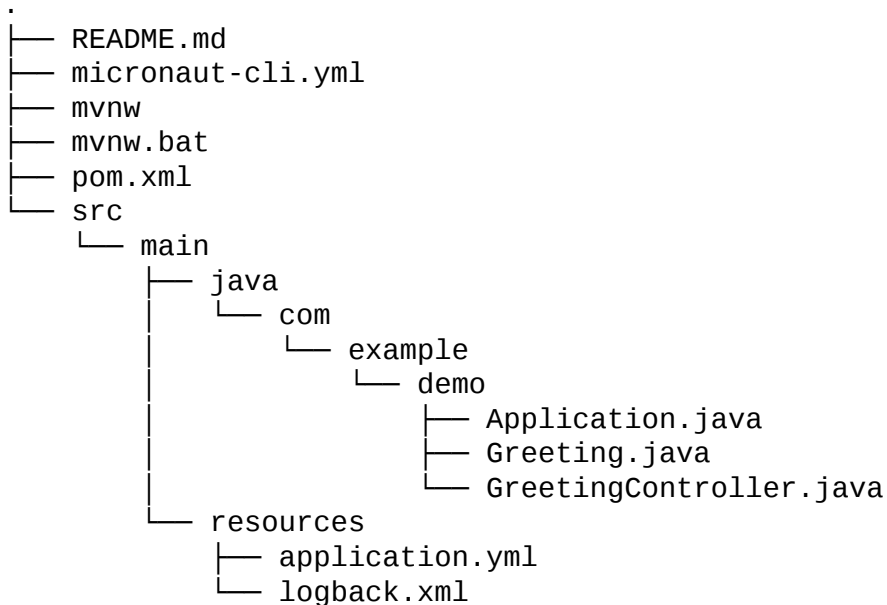
Micronaut

Micronaut began life in 2017 as a reimagination of the Grails framework but with a modern look. What is Grails you ask? It's one of the few successful "clones" of the Ruby on Rails (RoR for short) framework, leveraging the Groovy programming language. While Grails made quite the splash for a few years the rise of Spring Boot took it out of the spotlight prompting the Grails team to find alternatives which resulted in Micronaut. On the surface, Micronaut provides similar user experience as Spring Boot as it also allows developers to compose applications based on existing components and sensible defaults. But one of its key differentiators is the use of compile time Dependency Injection for assembling the application as opposed to runtime dependency injection which is the preferred way of assembling applications with Spring Boot, so far. This seemingly trivial change lets Micronaut exchange a bit of development time for a speed boost at runtime as the application spends less time bootstrapping itself; this can also lead to less memory consumption and less reliance on Java reflection which historically has been slower than direct method invocations.

There are a handful of ways to bootstrap a Micronaut project however the preferred one is to browse to <https://micronaut.io/launch> and select the settings and features you'd like to see added to the project. The default application type defines the minimum settings to build a REST based application such as the one we'll go through in a few minutes. Once satisfied with your selection you'd click on the "Generate Project" button which creates a Zip file that can be downloaded to your local development environment.

micronaut

Similarly as we did for Spring boot, unpacking the zip file and running the `./mvnw verify` command at the root directory of the project ensures that we have a sound starting point. This command invocation will download plugins and dependencies as needed; the build should succeed after a few seconds if everything goes right. The project structure should look like the following one after adding a pair of additional source files



The *Application.java* source file defines the entry point which we'll leave untouched for now as there's no need to make any updates. Similarly we'll leave the *application.yml* resource file unchanged as well; this resource supplies configuration properties which we don't require changes at this point. We need two additional source files: the data object defined by *Greeting.java* whose responsibility is to contain a message sent back to the consumer, and the actual REST endpoint defined by *GreetingController.java*. The controller stereotype goes all the way back to the conventions laid out by Grails, also followed by pretty much every RoR clone. You can certainly change the file name to anything that suits your domain, though you must leave the `@Controller` annotation in place. The source for the data object should look like this:

```
package com.example.demo;

import io.micronaut.core.annotation.Introspected;
```

```

@Intropected
public class Greeting {
    private final String content;

    public Greeting(String content) {
        this.content = content;
    }

    public String getContent() {
        return content;
    }
}

```

Once more we rely on an immutable design for this class. Note the use of the `@Intropected` annotation, this signals Micronaut to inspect the type at compile time and include it as part of the Dependency Injection procedure. Usually the annotation can be left out as Micronaut will figure out that the class is required, but its use is paramount when it comes to generating the native executable with GraalVM Native image otherwise the executable won't be complete. The second file should look like this:

```

package com.example.demo;

import io.micronaut.http.annotation.Controller;
import io.micronaut.http.annotation.Get;
import io.micronaut.http.annotation.QueryValue;

@Controller("/")
public class GreetingController {
    private static final String template = "Hello, %s!";

    @Get(uri = "/greeting")
    public Greeting greeting(@QueryValue(value = "name",
        defaultValue = "World") String name) {
        return new Greeting(String.format(template, name));
    }
}

```

We can appreciate that the controller defines a single endpoint mapped to `/greeting`, takes an optional parameter named `name` and returns an instance of the data object. By default Micronaut will marshall the return value as JSON thus there's no extra configuration required to make it happen. Now, running the application can be done in a couple of ways, you can either invoke `./mvnw`

`mn:run` which runs the application as part of the build process or invoke `./mvnw package` which creates a `demo-0.1.jar` in the `target` directory that can be launched in the conventional way, that is, `java -jar target/demo-0.1.jar`. Invoking a couple of queries to the REST endpoint may result in output similar to this one:

```
// using the default name parameter
$ curl http://localhost:8080/greeting
{"content":"Hello, World!"}

// using an explicit value for the name parameter
$ curl http://localhost:8080/greeting?name=Microservices
{"content":"Hello, Microservices!"}
```

Either command launches the application quite quickly. On my local environment the application is ready to process requests by 500ms on average, that's 3 times the speed of Spring Boot for equivalent behavior. The size of the JAR file is also a bit smaller at 14 MB in total. As impressive as these numbers may be, we can get a speed boost if the application were to be transformed using GraalVM Native image into a native executable. Fortunately for us the Micronaut way is friendlier with this kind of setup, resulting in everything we require already configured in the generated project. That's it, no need to update the build file with additional settings, it's all there. You do however require an installation of GraalVM and its `native-image` executable like we did before. Creating a native executable is as simple as invoking `./mvnw -Dpackaging=native-image package`, and after a few minutes we should get an executable named `demo` (as a matter of fact it's the project's `artifactId` if you were wondering) inside the `target` directory. Launching the application with the native executable results on 20ms startup time on average which is 1/3 speed gain compared to Spring Boot. The executable size is 60 MB which correlates to the reduced size of the JAR file.

We stop exploring Micronaut at this point and move to the next framework: Quarkus.

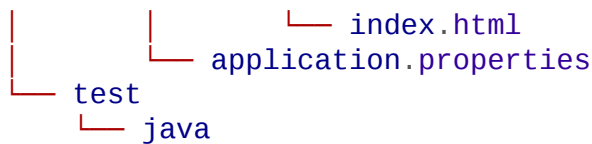
Quarkus

Although Quarkus was announced in early 2019 work began much earlier. Quarkus has a lot of similarities with the past two candidates, that is, it offers

Quarkus has a lot of similarities with the past two candidates, that is, it offers great development experience based on components, convention over configuration, and productivity tools. Even more, Quarkus decided to also use compile time Dependency Injection like Micronaut does, allowing it to reap the same benefits such as smaller binaries, faster startup, less runtime magic. But at the same time Quarkus adds its own flavor and distinctiveness and perhaps most importantly for some developers, Quarkus relies more on standards than the other two candidates. Quarkus implements the MicroProfile specifications which in turn are standards coming from JakartaEE (previously known as JavaEE) and additional standards developed under the MicroProfile project umbrella.

Getting started with Quarkus can be done in a similar fashion as we have seen so far with the previous frameworks, that is, you browse to a particular page to configure some values and download a Zip file. For Quarkus that page is <https://code.quarkus.io/> and it's loaded with plenty of goodies: there are many extensions to choose from to configure specific integrations such as databases, REST capabilities, monitoring, etc. Now to cover our specific use case we must select the RESTEasy Jackson extension, allowing Quarkus to seamlessly marshal values to and from JSON. Clicking on the “Generate your application” button should prompt you to save a Zip file into your local system, the contents of which should look similar to this:

```
.
├── README.md
├── mvnw
├── mvnw.cmd
├── pom.xml
├── src
│   ├── main
│   │   ├── docker
│   │   │   ├── Dockerfile.jvm
│   │   │   ├── Dockerfile.legacy-jar
│   │   │   ├── Dockerfile.native
│   │   │   └── Dockerfile.native-distless
│   │   ├── java
│   │   │   └── com
│   │   │       ├── example
│   │   │       │   └── demo
│   │   │       │       ├── Greeting.java
│   │   │       │       └── GreetingResource.java
│   │   └── resources
│   │       ├── META-INF
│   │       └── resources
```



We can appreciate that Quarkus adds Docker configuration files out of the box, reason being is that Quarkus was specifically designed to tackle microservice architectures in the cloud via containers and Kubernetes, however as time passed by its range has grown wider by supporting additional application types and architectures. The *GreetingResource.java* is also created by default and it's a typical JAX-RS resource. We'll have to make some adjustments to that resource to enable it to handle the *Greeting.java* data object whose source is displayed next:

```
package com.example.demo;

public class Greeting {
    private final String content;

    public Greeting(String content) {
        this.content = content;
    }

    public String getContent() {
        return content;
    }
}
```

The code is pretty much identical to what we've seen before in this chapter, there's nothing new nor surprising about this immutable data object. Now in the case of the JAX-RS resource things will look similar yet different, as the behavior we seek is the same however the way we instruct the framework to perform its magic is via JAX-RS annotations, thus the code looks like this:

```
package com.example.demo;

import javax.ws.rs.DefaultValue;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.QueryParam;

@Path("/greeting")
public class GreetingResource {
```

```

private static final String template = "Hello, %s!";

@GET
public Greeting greeting(@QueryParam("name")
    @DefaultValue("World") String name) {
    return new Greeting(String.format(template, name));
}
}

```

If you're familiar with JAX-RS then the code should be no surprise to you, but if you're not familiar with the JAX-RS annotations what we do here is mark the resource as with the REST path we'd like to react to; we also indicate that the `greeting()` method will handle a GET call, and that its `name` parameter has a default value. There's nothing more that needs to be done to instruct Quarkus to marshall the return value into JSON as that will happen by default.

Running the application can be done in a couple of ways as well, firstly using the developer mode as part of the build. This is one of the features that has a unique Quarkus flavor as it lets you run the application and pick up any changes you made automatically without having to manually restart the application. You can activate this mode by invoking the following command: `./mvnw compile quarkus:dev`. If you were to make any changes to the source files you'll notice that the build will automatically recompile and load the application. You may also run the application using the `java` interpreter as we've seen before which results in a command such as `java -jar target/quarkus-app/quarkus-run.jar`. Do note that we're using a different JAR although the `demo-1.0.0-SNAPSHOT.jar` does exist in the `target` directory; the reason to do it this way is that Quarkus applies custom logic to speed up the boot process even in the Java mode. Running the application should result in startup times with 600ms on average, pretty close to what Micronaut does. Also, the size of the full application is in the 13 MB range. Sending a couple of GET requests to the application without and with a `name` parameter result in output similar to the following one:

```

// using the default name parameter
$ curl http://localhost:8080/greeting
{"content":"Hello, World!"}

// using an explicit value for the name parameter
$ curl http://localhost:8080/greeting?name=Microservices
{"content":"Hello, Microservices!"}

```

```
{ "content": "Hello, Microservices!" }
```

It should be no surprise that Quarkus also supports generating native executables via GraalVM Native Image given that it targets cloud environments where small binary size is recommended. Because of this Quarkus comes with batteries included just like Micronaut and generates everything you need from the get go, there's no need to update the build configuration to get started with native executables. As it has been the case with the other examples you must ensure that the current JDK points to a GraalVM distribution and that the `native-image` executable is found in your path. Once this step has been cleared all that is left is to package the application as a native executable by invoking the following command: `./mvnw -Pnative package`. This activates the `native` profile which instructs the Quarkus build tools to generate the native executable.

After a couple of minutes the build should have produced an executable named *demo-1.0.0-SNAPSHOT-runner* inside the *target* directory. Running this executable shows that the application startups in 15ms on average! The size of the executable is close to 47 MB which makes Quarkus the framework that yields the fastest startup and smaller executable size so far when compared to previous candidate frameworks, but just for a bit.

We're done with Quarkus for the time being, leaving us with the fourth candidate framework: Helidon.

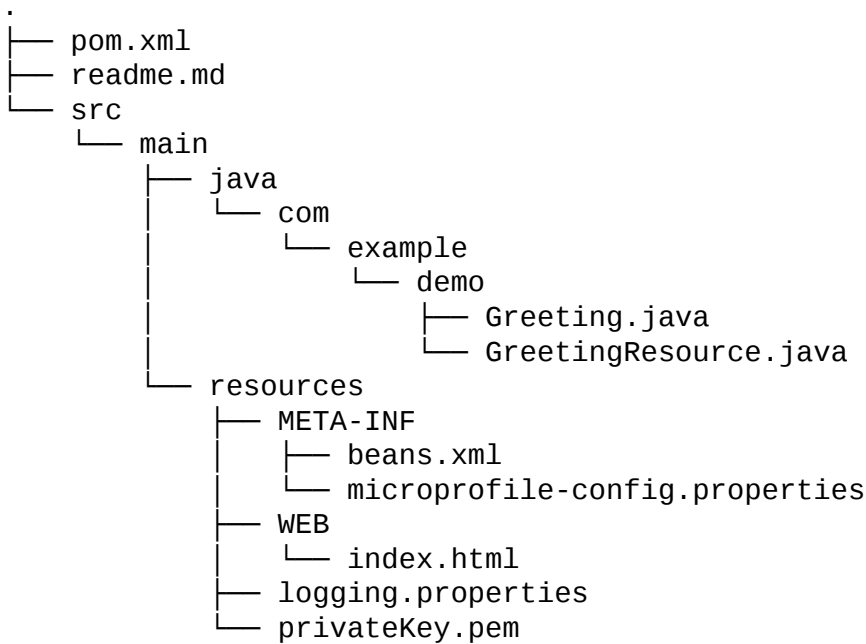
Helidon

Last but not least, Helidon is a framework specifically crafted for building microservices with two flavors: SE and MP. The MP flavor stands for MicroProfile and lets you build applications by harnessing the power of standards; this flavor is a full implementation of the MicroProfile specifications. The SE flavor on the other hand does not implement MicroProfile yet it delivers similar functionality to its MP counterpart albeit using a different set of APIs. You'd pick one flavor over the other depending on the APIs you'd like to interact with and a preference for standards or not; either way Helidon gets the job done.

Given that Helidon implements MicroProfile we can use yet another site to

bootstrap a Helidon project. This site is <https://start.microprofile.io> and can be used to create projects for all supported implementations of the MicroProfile specification by versions. You simply browse to the site, select which MP version you're interested in, select the MP implementation (in our case Helidon), and perhaps customize some of the available features, then click the "Download" button to download a Zip file containing the generated project. The Zip file contains a project structure similar to the following one except of course I've already updated the sources with the two files required to make the application work as we want it.

helidon



Alright, as it happens the source files *Greeting.java* and *GreetingResource.java* are identical to the sources we saw in the Quarkus example. How is that possible? Firstly because the code is definitely trivial but secondly and more importantly because both frameworks rely on the power of standards. Matter of fact the *Greeting.java* file is pretty much identical across all frameworks except for Micronaut where an additional annotation is needed, and only if you're interested in generating native executables, otherwise it's 100% identical. But in any case you decided to jump ahead to this section before browsing the others here's how the *Greeting.java* file looks like

```
package com.example.demo;

import io.helidon.common.Reflected;

@Reflected
public class Greeting {
    private final String content;

    public Greeting(String content) {
        this.content = content;
    }

    public String getContent() {
        return content;
    }
}
```

```
}  
}
```

Just a regular immutable data object with a single accessor. The *GreetingResource.java* file which defines the REST mappings needed for the application follows:

```
package com.example.demo;  
  
import javax.ws.rs.DefaultValue;  
import javax.ws.rs.GET;  
import javax.ws.rs.Path;  
import javax.ws.rs.QueryParam;  
  
@Path("/greeting")  
public class GreetingResource {  
    private static final String template = "Hello, %s!";  
  
    @GET  
    public Greeting greeting(@QueryParam("name")  
        @DefaultValue("World") String name) {  
        return new Greeting(String.format(template, name));  
    }  
}
```

We can appreciate the use of JAX-RS annotations, as we can see there's no need for Helidon specific APIs at this point. The preferred way to run a Helidon application is to package the binaries and run them with the `java` interpreter, that is, we lose a bit of build tool integration (for now) yet we can still use the command line to perform iterative development. Thus invoking `mvn package` followed by `java -jar /demo.jar` compiles, packages, and runs the application with an embedded web server listening on port 8080. We can send a couple of queries to it, such as

```
// using the default name parameter  
$ curl http://localhost:8080/greeting  
{"content":"Hello, World!"}  
  
// using an explicit value for the name parameter  
$ curl http://localhost:8080/greeting?name=Microservices  
{"content":"Hello, Microservices!"}
```

If you look at the output where the application process is running you'll see that

the application starts with 2.3s on average which puts it as the slowest candidate we have seen so far while the binaries size is close to 15 MB, putting it at the middle of all measurements. However as the adage goes “do not judge a book by its cover” Helidon does provide more features out of the box automatically configured which would account for the extra startup time and the larger deployment size.

If startup speed and deployment size were to be an issue one could reconfigure the build to remove those features that may not be needed, as well as switching to native executable mode. Fortunately the Helidon team has embraced GraalVM Native image as well and every Helidon project bootstrapped as we’ve done ourselves comes with the required configuration to create native binaries, there’s no need to tweak the *pom.xml* file if we follow the conventions. Simply execute the `mvn -Pnative-image package` command and you’ll find a binary executable named *demo* inside the *target* directory. This executable weights about 94 MB, the largest so far, while it’s startup time is 50ms on average which is in the same range as the previous frameworks.

Up to now we’ve caught a glimpse of what each framework has to offer, from base features to build tool integration. As a reminder there are several reasons to pick one candidate framework over another. I encourage you to write down a matrix for each relevant feature/aspect that affects your development requirements and assess each one of those items with every candidate.

Serverless

We began this chapter by looking at what monolithic applications and architectures, usually piece together by components and tiers clumped together into a single, cohesive unit. Changes or updates to a particular piece require updating and deploying the whole. Failure at one particular place could bring down the whole as well. To mitigate these issues we moved on, into microservices. Breaking down the monolith into smaller chunks that can be updated and deployed individually and independently from one another should take care of the previously mentioned issues but brings a host of other issues.

Before it was enough to run the monolith inside an application server hosted on big iron, with a handful of replicas and a load balancer for good measure. This setup has scalability issues. With the microservices approach we can grow or

setup has scalability issues. With the microservices approach we can grow or collapse the mesh of services depending on the load, this boosts elasticity but now we have to coordinate multiple instances, provisioning runtime environment, load balancers become a must, API gateways make an appearance, network latency rears its ugly face, and did i mention distributed tracing? Yes, that's a lot of things that one has to be aware of and manage. Then again what if you didn't have to? What if someone else takes care of the infrastructure, monitoring, and other "minutiae" required to run applications at scale? This is where the serverless approach comes in. The promise is to concentrate on the business logic at hand and let the serverless provider deal with everything else.

While distilling a component into smaller pieces one thought should have come to mind: what's the smallest reusable piece of code I can turn this component into? if your answer is a Java class with a handful of methods and perhaps a couple of injected collaborators/services then you're very close but not there yet. The smallest piece of reusable code is as a matter of fact, a single method. Picture for a moment the following: there is a microservice defined as a single class which performs the following steps:

- read the input arguments and transform them into a consumable format as required by the next step.
- perform the actual behavior required by the service, such as issuing a query to a database, indexing, logging, or else.
- transform the processed data into an output format.

Now, each one of these steps may be organized in separate methods. You may soon realize that some of these methods are reusable as is or parameterized. A typical way to solve this would be to provide a common super type among microservices. This creates a strong dependency between types, and for some use cases that's alright, but for others where updates to the common code have to happen as soon as possible, in a versioned fashion, without disrupting currently running code then I'm afraid we may need an alternative. With this scenario in mind, if the common code were to be provided instead as a set of methods that can be invoked independently from one another, with their inputs and outputs composed in such way that you establish a pipeline of data transformations, then we arrive at what is now known as functions, with offerings such as "function-as-a-service" (or FaaS for short) being a common subject among serverless

providers. In summary, FaaS is a fancy way to say you compose applications based on the smallest deployment unit possible and let the provider figure out all the infrastructure details for you. In the following sections we'll build and deploy a simple function to the cloud.

Setting Up

Nowadays every major cloud provider has a FaaS offering at your disposal, with addons that hook into other tools for monitoring, logging, disaster recovery, and more; just pick the one that meets your needs. For the sake of this chapter we'll pick Amazon Lambda, after all it's the originator of the FaaS idea. We'll also pick Quarkus as the implementation framework, as it's the one that currently provides the smallest deployment size. Be aware that the configuration shown here may need some tweaks or be totally outdated; always review the latest versions of the tools required to build and run the code. We'll use Quarkus 1.13.7 for now.

Setting up a function with Quarkus and Amazon Lambda requires having an [Amazon AWS account](#), the [AWS CLI](#) installed on your system, and the [AWS SAM CLI](#) if you'd like to run local tests. Once you have this covered the next step is to bootstrap the project for which we would inclined to use <https://code.quarkus.io/> as we did before except that a function project requires different setup, thus it's better if we switch to using a Maven Archetype, such that

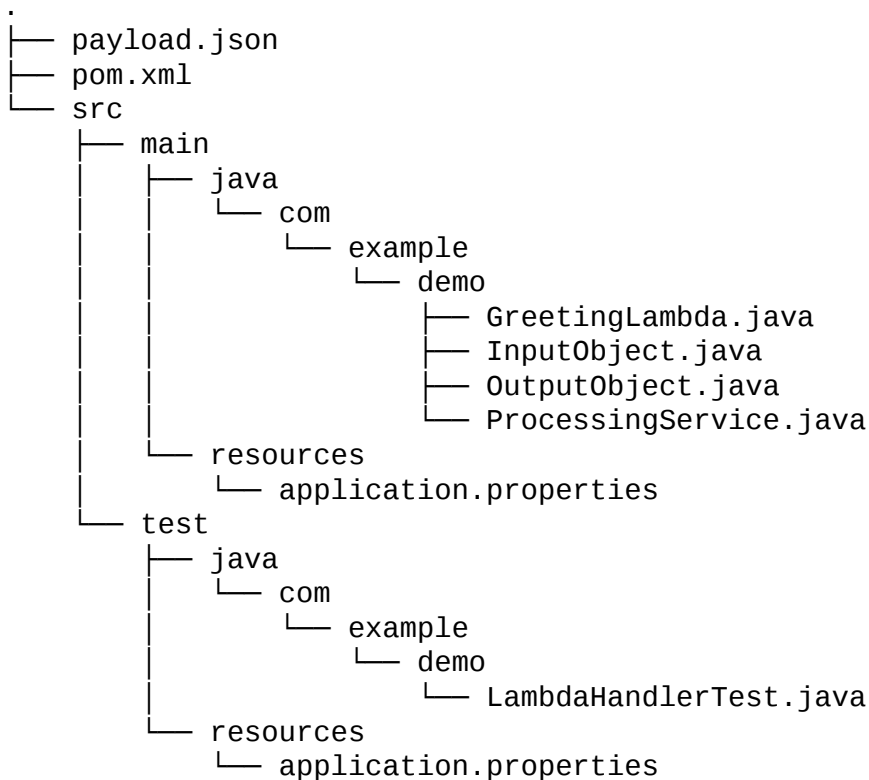
```
mvn archetype:generate \
  -DarchetypeGroupId=io.quarkus \
  -DarchetypeArtifactId=quarkus-amazon-lambda-archetype \
  -DarchetypeVersion=1.13.7.Final
```

Invoking this command in interactive mode will ask you a few questions such as the GAV coordinates for the project and the base package. For this demo let's go with:

- groupId: com.example.demo
- artifactId: demo
- version: 1.0-SNAPSHOT (the default)

- package: com.example.demo (same as groupId)

This results in a project structure suitable to build, test, and deploy a Quarkus project as a function deployable to Amazon Lambda. The archetype creates build files for both Maven and Gradle but we don't need the latter for now; it also creates 3 function classes but we only need one. Our aim is to have a file structure similar to this one



The gist of the function is to capture inputs with the `InputObject` type, process them with the `ProcessingService` type, then transform the results into another type (`OutputObject`). The `GreetingLambda` type ties everything together. Let's have a look at both input and output types first, after all they are simple types that are only concerned with holding data, no logic whatsoever:

```
package com.example.demo;

public class InputObject {
    private String name;
    private String greeting;
```

```

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getGreeting() {
        return greeting;
    }

    public void setGreeting(String greeting) {
        this.greeting = greeting;
    }
}

```

The lambda expects two input values: a greeting and a name. We'll see how they get transformed by the processing service in a moment.

```

package com.example.demo;

public class OutputObject {
    private String result;
    private String requestId;

    public String getResult() {
        return result;
    }

    public void setResult(String result) {
        this.result = result;
    }

    public String getRequestId() {
        return requestId;
    }

    public void setRequestId(String requestId) {
        this.requestId = requestId;
    }
}

```

The output object holds the transformed data and a reference to the request id. We'll use this field to show how we can get data from the running context. Alright, the processing service is next; this class is responsible for transforming

the inputs into outputs, in our case it concatenates both input values into a single String, as shown next:

```
package com.example.demo;

import javax.enterprise.context.ApplicationScoped;

@ApplicationScoped
public class ProcessingService {
    public OutputObject process(InputObject input) {
        OutputObject output = new OutputObject();
        output.setResult(input.getGreeting() + " " + input.getName());
        return output;
    }
}
```

What's left is to have a look at `GreetingLambda`, the type used to assemble the function itself. This class requires implementing a known interface supplied by Quarkus, whose dependency should be already configured in the *pom.xml* created with the archetype. This interface is parameterized with input and output types, luckily we have those already. Every lambda must have a unique name and may access its running context, as shown next:

```
package com.example.demo;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;

import javax.inject.Inject;
import javax.inject.Named;

@Named("greeting")
public class GreetingLambda implements RequestHandler<InputObject,
OutputObject> {
    @Inject
    ProcessingService service;

    @Override
    public OutputObject handleRequest(InputObject input, Context
context) {
        OutputObject output = service.process(input);
        output.setRequestId(context.getAwsRequestId());
        return output;
    }
}
```

All the pieces fall into place. The lambda defines input and output types and invokes the data processing service. For the purpose of demonstration this example shows the use of dependency injection, however one could reduce the code by moving the behavior of `ProcessingService` into `GreetingLambda`. we can quickly verify the code by running local tests with `mvn test`, or if you prefer `mvn verify` as that will also package the function. Note that additional files are placed in the *target* directory when the function is packaged, specifically a script named `manage.sh` which relies on the AWS CLI tool to create, update, and delete the function at the target destination associated with your AWS account. Additional files are required to support these operations:

- `function.zip` - the deployment file containing the binary bits.
- `sam.jvm.yaml` - local test with AWS SAM CLI (java mode).
- `sam.native.yaml` - local test with AWS SAM CLI (native mode).

The next step requires you to have an *Execution Role* configured for which it's best to refer to the [AWS Getting Started Guide](#) in case the procedure has been updated. The guide shows you how to get the AWS CLI configured (if you have not done so already) and create an execution role which must be added as an environment variable to your running shell, for example

```
LAMBDA_ROLE_ARN="arn:aws:iam::1234567890:role/lambda-ex"
```

In this case `1234567890` stands for your AWS account ID and `lambda-ex` is the name of the role of your choosing. We can proceed with executing the function for which we have two modes (Java, native) and two execution environments (local, production); let's tackle the Java mode first for both environments then follow it up with native mode. Running the function on a local environment requires the use of a Docker daemon which by now should be common place on a developer's toolbox; we also require using the AWS SAM CLI to drive the execution. Remember the set of additional files found inside the *target* directory? We'll make use of the `sam.jvm.yaml` file alongside another file that was created by the archetype when the project was bootstrapped, this file is named `payload.json` located at the root of the directory, its contents

should look like this

```
{
  "name": "Bill",
  "greeting": "hello"
}
```

This file defines values for the inputs accepted by the function. Given the function is already packaged we just have to invoke it like so

```
$ sam local invoke --template target/sam.jvm.yaml --event payload.json
Invoking
io.quarkus.amazon.lambda.runtime.QuarkusStreamHandler::handleRequest
(java11)
Decompressing /work/demo/target/function.zip
Skip pulling image and use local one: amazon/aws-sam-cli-emulation-
image-java11:rapid-1.24.1.
```

```
Mounting
/private/var/folders/p_/3h19jd792gq0zr1ckqn9jb0m0000gn/T/tmpptesjj0c8
as /var/task:ro,delegated inside runtime container
START RequestId: 0b8cf3de-6d0a-4e72-bf36-232af46145fa Version: $LATEST
```

```
--/___\// // // _ | /_ \// // // // _/
-/ // // // // _ | /, _/ ,< // // /\ \
--\___\___// _/ |/_/|/_/|_|\___/_/
[io.quarkus] (main) quarkus-lambda 1.0-SNAPSHOT on JVM (powered by
Quarkus 1.13.7.Final) started in 2.680s.
[io.quarkus] (main) Profile prod activated.
[io.quarkus] (main) Installed features: [amazon-lambda, cdi]
END RequestId: 0b8cf3de-6d0a-4e72-bf36-232af46145fa
REPORT RequestId: 0b8cf3de-6d0a-4e72-bf36-232af46145fa  Init Duration:
1.79 ms Duration: 3262.01 ms    Billed Duration: 3300 ms          Memory
Size: 256 MB    Max Memory Used: 256 MB
{"result":"hello Bill","requestId":"0b8cf3de-6d0a-4e72-bf36-
232af46145fa"}
```

The command will pull a Docker image suitable for running the function. Take note of the reported values which may differ depending on your setup. On my local environment this function would cost me 3.3s and 256 MB for its execution. This can give you an idea of how much you'll be billed when running your system as a set of functions, however local is not the same as production thus let's deploy the function to the real deal. We'll make use of the `manage.sh` script to accomplish this feat, by invoking the following commands:

```

$ sh target/manage.sh create
$ sh target/manage.sh invoke
Invoking function
++ aws lambda invoke response.txt --cli-binary-format raw-in-base64-out --function-name QuarkusLambda --payload file://payload.json --log-type Tail --query LogResult --output text
++ base64 --decode
START RequestId: df8d19ad-1e94-4bce-a54c-93b8c09361c7 Version: $LATEST
END RequestId: df8d19ad-1e94-4bce-a54c-93b8c09361c7
REPORT RequestId: df8d19ad-1e94-4bce-a54c-93b8c09361c7 Duration: 273.47 ms Billed Duration: 274 ms Memory Size: 256 MB Max Memory Used: 123 MB Init Duration: 1635.69 ms
{"result":"hello Bill","requestId":"df8d19ad-1e94-4bce-a54c-93b8c09361c7"}

```

As we can appreciate the billed duration and memory usage decreased which is good for our wallet although the init duration went up to 1.6 which would delay the response, increasing the total execution time across the system. Let's see how these numbers change when we switch from Java mode to native mode. As you may recall Quarkus let's you package projects as native executables out of the box but we must also remember that Amazon Lambda requires Linux executables, thus if you happen to be running on a non-Linux environment you'll need to tweak the packaging command. Here's what needs to be done

```

# for linux
$ mvn -Pnative package

# for non-linux
$mvn package -Pnative -Dquarkus.native.container-build=true -Dquarkus.native.container-runtime=docker

```

The second command invokes the build inside a Docker container and places the generated executable on your system at the expected location while the first command executes the build as is. With the native executable now in place we can execute the new function both in local and production environments. Let's see the local environment first:

```

$ sam local invoke --template target/sam.native.yaml --event payload.json
Invoking not.used.in.provided.runtime (provided)
Decompressing /work/demo/target/function.zip
Skip pulling image and use local one: amazon/aws-sam-cli-emulation-image-provided:rapid-1.24.1.

```

```

Mounting
/private/var/folders/p_/3h19jd792gq0zr1ckqn9jb0m0000gn/T/tmp1zgzkuyh
as /var/task:ro,delegated inside runtime container
START RequestId: 27531d6c-461b-45e6-92d3-644db6ec8df4 Version: $LATEST

--/ _ _ \ / / / / _ | / _ \ / / / / / _ /
-/ / / / / / / _ | / , _ / , < / / / \ \
--\ _ _ \ _ _ / / | / / | / / | \ _ _ / _ /
[io.quarkus] (main) quarkus-lambda 1.0-SNAPSHOT native (powered by
Quarkus 1.13.7.Final) started in 0.115s.
[io.quarkus] (main) Profile prod activated.
[io.quarkus] (main) Installed features: [amazon-lambda, cdi]
END RequestId: 27531d6c-461b-45e6-92d3-644db6ec8df4
REPORT RequestId: 27531d6c-461b-45e6-92d3-644db6ec8df4 Init Duration:
0.13 ms Duration: 218.76 ms Billed Duration: 300 ms Memory Size:
128 MB Max Memory Used: 128 MB
{"result":"hello Bill","requestId":"27531d6c-461b-45e6-92d3-
644db6ec8df4"}

```

The billed duration decreased by one order of magnitude, going from 3300ms to just 300ms, and the used memory was halved; this is looking promising compared from its Java counterpart. Will we bet better numbers when running on production? Let's have a look.

```

$ sh target/manage.sh native create
$ sh target/manage.sh native invoke
Invoking function
++ aws lambda invoke response.txt --cli-binary-format raw-in-base64-
out --function-name QuarkusLambdaNative --payload file://payload.json
--log-type Tail --query LogResult --output text
++ base64 --decode
START RequestId: 19575cd3-3220-405b-afa0-76aa52e7a8b5 Version: $LATEST
END RequestId: 19575cd3-3220-405b-afa0-76aa52e7a8b5
REPORT RequestId: 19575cd3-3220-405b-afa0-76aa52e7a8b5 Duration: 2.55
ms Billed Duration: 187 ms Memory Size: 256 MB Max Memory
Used: 54 MB Init Duration: 183.91 ms
{"result":"hello Bill","requestId":"19575cd3-3220-405b-afa0-
76aa52e7a8b5"}

```

The total billed duration results in 30% speed up and the memory usage is less than half than before; however the real winner is the initialization time which takes roughly 10% of the previous time. This means running your function in native mode results in faster startup and better numbers across the board. Now it's up to you to decide the combination of options that will give you the best results. Sometimes staying on the Java mode is good enough even for production

results, sometimes staying on the Java mode is good enough even for production, or going native all the way is what may give you the edge, whichever way it may be measurements are key, don't guess!

Conclusion

We covered a lot of ground in this chapter, starting with a traditional monolith, breaking it apart into smaller parts with reusable components that can be deployed independently from one another, known as microservices, going all the way to the smallest deployment unit possible: a function. There are trade-offs along the way as microservice architectures are inherently more complex as they are composed of more moving parts, network latency becomes a real issue and must be tackled accordingly, other aspects such as data transactions become more complex as their span may cross service boundaries depending on the case. The use of Java and native executable mode yields different results and requires customized setup, each with their own pros and cons. My recommendation dear reader is to evaluate, measure, then select a combination; keep tabs on numbers and SLAs, you may need to re-evaluate decisions along the road and make adjustments.

The following table summarizes the measurements obtained by running the sample application on both Java and native image modes, on my local environment and remote, for each one of the candidate frameworks. The size columns show the deployment unit size, while the time columns depict the time from startup up to the first request.

| Framework | Java - size | Java - time | Native- size | Native - time |
|-------------|-------------|-------------|--------------|---------------|
| Spring Boot | 17 MB | 2200ms | 78 MB | 90ms |
| Micronaut | 14 MB | 500ms | 60 MB | 20ms |
| Quarkus | 13 MB | 600ms | 47 MB | 13ms |
| Helidon | 15 MB | 2300ms | 94 MB | 50ms |

As a reminder, you are encouraged to take your own measurements, changes to the hosting environment, JVM version & settings, framework version, network conditions, and other environment characteristics will yield different results. The

conditions, and other environment characteristics will yield different results. The numbers shown before should be taken with a grain salt, never as authoritative values.

-
- 1 Alexander of Aphrodisias, On Aristotle 'Prior Analytics' 1.1-7, tr. J. Barnes et al., London: Duckworth, 1991
 - 2 Software development, Wikipedia, https://en.wikipedia.org/wiki/Software_development
 - 3 Peter Mell (NIST), Tim Grance (NIST). The NIST Definition of Cloud Computing. September 2011
 - 4 Canalys "Global cloud infrastructure market Q4 2020 " <https://www.canalys.com/newsroom/global-cloud-market-q4-2020>
 - 5 Canalys "Global cloud infrastructure market Q4 2020 " <https://www.canalys.com/newsroom/global-cloud-market-q4-2020>
 - 6 Kilcioglu, C. Rao, J.M. Kannan, A. and McAfee, R.P. Usage patterns and the economics of the public cloud. In Proceedings of the 26th International Conference World Wide Web, 2017
 - 7 Rodgers, Peter. "Service-Oriented Development on NetKernel- Patterns, Processes & Products to Reduce System Complexity Web Services Edge 2005 East: CS-3". CloudComputingExpo 2005. SYS-CON TV. Archived from the original on 20 May 2018. Retrieved 3 July 2017.
 - 8 Chris Richardson. "Pattern: Microservice Architecture" <https://microservices.io/patterns/microservices.html>
 - 9 Mark Richards. "Microservices AntiPatterns and Pitfalls," O'Reilly
 - 10 D. Taibi, V. Lenarduzzi. "On the Definition of Microservice Bad Smells" IEEE Software. Vol 35, Issue 3, May/June 2018

Chapter 4. Continuous Integration

Melissa McKay

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors’ raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 5th chapter of the final book. If there is a GitHub repo associated with the book, it will be made active after final publication.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at ccollins@oreilly.com.

Always make new mistakes.

—Esther Dyson

In Chapter 4, you learned the value of source control and a common code repository. Now that you have organized and settled on your source control solution, what next? Simply having a common location where all developers on a project can update and maintain the code base is a long way from getting your software deployed to a production environment. There are a few more steps you must take to get the end result to a place where your users can bask in the perfect user experience of your delivered software.

Think about the process that you would take as an individual developer to progress your software through the entire software development lifecycle. After determining the acceptance criteria for a particular feature or bug fix for your software, you would proceed with adding the actual lines of code along with the related unit tests to the codebase. Then, you would compile and run all of the unit tests to ensure that your new code works like you expect, (or at least as defined by your unit tests), and doesn’t break known existing functionality. After you find that all tests pass, you would build and package your application and verify functionality in the form of integration tests in a Quality Assurance (QA)

verify functionality in the form of integration tests in a Quality Assurance (QA) environment. Finally, happy with the green light from your well oiled and maintained test suites, you would deliver and/or deploy your software to a production environment.

If you have any development experience at all, you know as well as I do that software rarely falls into place so neatly. Strict implementation of the ideal workflow described is too simplistic when you begin working on a larger project with a team of developers. There are a number of complications introduced that can gum up the gears of the software delivery lifecycle and throw your schedule into a lurch. This chapter will discuss how Continuous Integration (CI) and the related best practices and toolsets will help you steer clear of or mitigate the most common hurdles and headaches that software development projects often encounter on the path to delivery.

Adopt Continuous Integration

Continuous Integration, or CI, is most simply described as frequently integrating code changes from multiple contributors into the main source code repository of a project. In practice, this definition by itself is a little vague. Exactly how often is implied by frequently? What does integrating actually mean in this context? Is it enough just to coordinate pushing code changes to the source code repository? And most importantly, what problem does this process solve - for what benefit(s) should I adopt this practice?

The concept of CI has been around now for quite some time. According to Martin Fowler, the term *Continuous Integration* originated with Kent Beck's Extreme Programming development process, as one of its original twelve practices.¹ In the DevOps community, the term itself is now as common as butter on toast. But how it is implemented may vary from team to team and project to project. The benefits are hit or miss if there isn't a thorough understanding of the original intent or if best practices are abandoned.

CI is meant to identify bugs and compatibility issues as quickly as possible in the development cycle. The basic premise of CI is that if developers integrate changes often, bugs can be found sooner in the process and less time is spent hunting down when and where a problem was introduced. The longer a bug goes undiscovered, the more potential there is for it to become entrenched in the

surrounding code base. It is much easier from a development perspective to find, catch and fix bugs closer to when they are introduced rather than to extract them from layers of code that have already moved to later stages of the delivery pipeline. Bugs that evade discovery until the latest acceptance phases, and especially those that escape all the way to release, directly translate to more money to fix and less time to spend on new features. In the case of fixing a bug in production, in many instances, there is now a requirement to patch existing deployments in addition to including and documenting the fix in a new version. This inherently reduces the time the team has available to spend on the development of new features.

It's important to understand that implementing a CI solution does *not* equate to software that never has any bugs. It would be foolish to use such a definitive measure to determine whether the implementation of CI is worthy. A more valuable metric might be the number of bugs or compatibility issues that were caught by CI. In much the same way that a vaccine is never 100% effective in a large population, CI is simply another level of protection to filter the most obvious bugs from a release. By itself, CI will never replace the well-known benefits of software development best practices that are in the purview of the initial design and development steps. It will, however, provide a safety net for software as it is repeatedly handled and massaged by multiple developers over time.

Continuous Integration doesn't get rid of bugs, but it does make them dramatically easier to find and remove.

—Martin Fowler

My first experience with CI was during an internship at a small company that adopted the software development methodology of Extreme Programming[], of which CI is an important aspect.² We did not have an incredibly fancy system using all of the latest and greatest DevOps tools. What we did have in place was a common code repository, and a single build server located in a small closet in the office. Unbeknownst to me when I first joined the development team, there was also a speaker set up on the build server that would erupt in the sound of emergency sirens if a fresh check-out from source control resulted in the failure of a build or any automated tests. We were a relatively young team, so this part of our CI was mostly in jest, but guess who learned *remarkably* quickly not to

push code to the main repository without first verifying that the project built successfully and passed unit tests? To this day, I feel exceptionally fortunate to have been exposed to this practice in this way. The simplicity of it underscored the most important aspects of CI.

There were three byproducts of this simple setup that I want to call out:

1. Code integration was regular and rarely complicated.

My team had agreed to follow XP practices, where integration is encouraged as often as every few hours.³ More important than a specific time interval, was the amount of code that required integration at any given point. When planning and breaking down actual development work, we focused our efforts on creating small, completable tasks, always beginning with the simplest thing that can possibly work.⁴ By “completable”, I mean after the development task was complete, it could be integrated into the main code repository and the result would be expected to build successfully and pass all unit tests. This practice of organizing code updates in as small a package as possible made regular and frequent integration to the main source code repository a normal and unremarkable activity. Rarely was there significant time spent on large integration efforts.

2. Build and test failures were relatively easy to troubleshoot.

Because the project was built and automated tests were run at regular intervals, it was readily apparent where to start troubleshooting any failures. A relatively small amount of code would have been touched since the latest successful build, and if the problem couldn't immediately be identified and resolved, we would start with reverting the latest merge and work backward as needed in order to restore a clean build.

3. Bugs and compatibility issues introduced by integration and caught by the CI system were fixed immediately.

The loud sound of the siren let *everyone* on the team know there was a problem that needed to be addressed, a problem that could not be ignored. Because our CI system halted progress whenever there was a

build or test failure, everyone was on board to figure out what was wrong and what to do to fix the problem. Team communication, coordination, and cooperation were all in top shape because no-one would be able to move forward until the issue was resolved. A majority of the time, the offending code could be identified simply by analyzing the most recent merge and the responsibility to fix was assigned to that developer or pair of developers. There were also times when a discussion with the entire team was necessary because of a compatibility issue around multiple recent merges where changes in one part of the system negatively affected another seemingly unrelated part. These instances required our team to re-evaluate the code changes being made holistically and then decide together the best plan of action.

These three things were key to the success of our CI solution. You might have discerned that all three of these aspects imply the prerequisites of a healthy code base and a healthy development team. Without these, the initial implementation of a CI solution will undoubtedly be more difficult. However, implementing a CI solution will in turn have a positive impact on the codebase as well as the health of the team, and taking the first steps will provide a measure of benefit that will be well worth the effort. It is true there is much more to an effective CI solution than simple coordination of code contribution to a shared repository and following a mandate to integrate at an agreed-upon frequency. The following sections will walk you through the essentials of a complete, practicable, CI solution that will help to unburden and accelerate the software development process.

Declaratively Script Your Build

These three things were key to the success of our CI solution. You might have discerned that all three of these aspects imply the prerequisites of a healthy code base and a healthy development team. Without these, the initial implementation of a CI solution will undoubtedly be more difficult. However, implementing a CI solution will in turn have a positive impact on the codebase as well as the health of the team, and taking the first steps will provide a measure of benefit that will be well worth the effort. It is true there is much more to an effective CI solution than simple coordination of code contribution to a shared repository and

following a mandate to integrate at an agreed-upon frequency. The following sections will walk you through the essentials of a complete, practicable, CI solution that will help to unburden and accelerate the software development process.

You will reap a tremendous amount of time savings from taking this step alone. Your project *build lifecycle* (all of the discrete steps required to build your project) can easily grow more complicated over time, especially as you consume more and more dependencies, include various resources, add modules, and add tests. You may also need to build your project differently depending on the intended deployment environment. For example, you might need to enable debugging capabilities in a development or quality assurance environment, but disable debugging in a build intended for release to production as well as prevent test classes from being included in the distributable package. Manually performing all of the required steps involved in building a java project, including consideration for configuration differences per environment, is a hotbed for human error. The first time you neglect a step like building an updated dependency and consequently must repeat a build of a huge multi-module project to correct your mistake, you will appreciate the value of a build script.

Whatever tool or framework you choose for scripting your build, take care to use a *declarative* approach rather than *imperative*. A quick reminder of the meaning of these terms:

Imperative

Defining an exact procedure with implementation details

Declarative

Defining an action without implementation details

In other words, keep your build script focused on *what* you need to do rather than *how* to do it. This will help keep your script understandable, maintainable, testable, and scalable by encouraging reuse on other projects or modules. To accomplish this, you may need to establish or conform to a known convention, or write plugins or other external code referenced from your build script that provides the implementation details. Some build tools are more apt to foster a declarative approach than others. This usually comes with the cost of

conforming to a convention versus flexibility.

The java ecosystem has several well-established build tools available, so I would be surprised if you are currently manually compiling your project with javac and packaging your class files into a jar or other package type. You likely already have some sort of build process and script established, but in the unlikely scenario that you do not, you are starting a brand-new java project, or you are looking to improve an existing script to utilize best practices, this section will summarize a few of the most common build tools/frameworks available in the Java ecosystem and what they provide you out of the box. First, it is important to map out your build process in order to determine what you need from your build script in order to gain the most benefit.

To build a java project, at the bare minimum you will need to specify the following:

Java version

The version of Java required to compile the project

Source directory path

The directory that includes all of the source code for the project

Destination directory path

The directory where compiled class files are expected to be placed

Names, locations, and versions of needed dependencies

The metadata necessary to locate and gather any dependencies required by your project

With this information, you should be able to execute a minimal build process with the following steps:

1. Collect any needed dependencies.
2. Compile the code.
3. Run tests.

4. Package your application.

The best way to show how to massage your build process into a build script is by example. The following examples demonstrate the use of three of the most common build tools to script the minimal build process described for a simple Hello World Java application. In no way do these examples explore all of the functionality available in these tools. They are simply meant as a crash course to help you either begin to understand your existing build script or help you write your first build script to benefit from a full CI solution.

In evaluating a build tool, bear in mind the actual process your project requires to complete a build. It may be that your project requires scripting of additional steps that are not shown here and one build tool may be more suited than another to accomplish this. It is important that the tool you choose helps you programmatically define and accelerate the build process your project requires rather than arbitrarily force you to modify your process to fit the requirements of the tool. That said, when you learn the capabilities of a tool, reflect on your process and be mindful of changes that would benefit your team. This is most important with established projects. Changes to the process, however well-intentioned, can be painful for a development team. They should only be made intentionally, with a clear understanding of the reason for the change and of course, a clear benefit.

Build With Apache Ant

Apache Ant is an open-source project released under an Apache License by the Apache Software Foundation. According to the [Apache Ant documentation](#) the name is an acronym for “Another Neat Tool” and was initially part of the Tomcat codebase, written by James Duncan Davidson for the purpose of building Tomcat. It’s first initial release was on July 19, 2000.

Apache Ant is a build tool written in Java that provides a way to describe a build process as declarative steps within an XML file. This is the first build tool that I was exposed to in my Java career and although Ant has heavy competition today, it is still an active project and widely used often in combination with other tools.

KEY ANT TERMINOLOGY

Ant Task

An Ant task is a small unit of work such as deleting a directory or copying a file. Under the covers, Ant tasks map to Java objects which contain the implementation details for the task. There are a large number of built-in tasks available in Ant as well as the ability to create custom tasks.

Ant Target

Ant tasks are grouped into Ant targets. An Ant target is invoked by Ant directly. For example, for a target named *compile*, you would run the command `ant compile`. Ant targets can be configured to depend on each other in order to control the order of execution.

LISTING ANT TARGETS

Some Ant build files can grow to be quite large. In the same directory as *build.xml*, you can run the following command to get a list of available targets:

```
ant -projecthelp
```

Ant Build File

The Ant build file is an XML file used to configure all of the Ant tasks and targets utilized by a project. By default, this file is named *build.xml* and is found at the root of the project directory.

Example 4-1 is a simple Ant build file I created and executed with Ant 1.10.8.

Example 4-1. Ant build script (build.xml)

```
<project name="my-app" basedir="." default="package"> ☐  
  
  <property name="version" value="1.0-SNAPSHOT"/> ☐  
  <property name="finalName" value="${ant.project.name}-${version}"/>  
  <property name="src.dir" value="src/main/java"/>
```



```

<property name="build.dir" value="target"/>
<property name="output.dir" value="${build.dir}/classes"/>
<property name="test.src.dir" value="src/test/java"/>
<property name="test.output.dir" value="${build.dir}/test-classes"/>
<property name="lib.dir" value="lib"/>

<path id="classpath"> ☐
    <fileset dir="${lib.dir}" includes="**/*.jar"/>
</path>

<target name="clean">
    <delete dir="${build.dir}"/>
</target>

<target name="compile" depends="clean"> ☐
    <mkdir dir="${output.dir}"/>
    <javac srcdir="${src.dir}"
        destdir="${output.dir}"
        target="11" source="11"
        classpathref="classpath"
        includeantruntime="false"/>
</target>

<target name="compile-test">
    <mkdir dir="${test.output.dir}"/>
    <javac srcdir="${test.src.dir}"
        destdir="${test.output.dir}"
        target="11" source="11"
        classpathref="classpath"
        includeantruntime="false"/>
</target>

<target name="test" depends="compile-test"> ☐
    <junit printsummary="yes" fork="true">
        <classpath>
            <path refid="classpath"/>
            <pathelement location="${output.dir}"/>
            <pathelement location="${test.output.dir}"/>
        </classpath>

        <batchtest>
            <fileset dir="${test.src.dir}"
includes="**/*Test.java"/>
        </batchtest>
    </junit>
</target>

<target name="package" depends="compile,test"> ☐
    <mkdir dir="${build.dir}"/>
    <jar jarfile="${build.dir}/${finalName}.jar"

```

```
        basedir="${output.dir}"/>
    </target>
</project>
```

- ☐ The value of the *default* attribute of the project can be set to the name of a default target to run when Ant is invoked without a target. For this project, the command `ant` without any arguments will run the *package* target.
- ☐ Property elements are hardcoded values that may be used more than once in the rest of the build script. Using them helps keep with both readability and maintainability.
- ☐ This path element is how I chose to manage the location of needed dependencies for this project. In this case, both the *junit* and *hamcrest-core* jars are manually placed in the directory configured here. This technique implies that dependencies would be checked into source control along with the project. Although it was simple to do for this project for this example, this is not a recommended practice. Chapter 6 will discuss package management in detail.
- ☐ The `compile` target is responsible for the compilation of the source code (this project specifies Java 11) and placement of the resulting class files in the configured location. This target depends on the *clean* target, meaning the `clean` target will be run first, to ensure that compiled class files are fresh and not leftover from an old build.
- ☐ The *test* target configures the `junit` Ant task which will run all of the available unit tests and print the results to the screen.
- ☐ The *package* target will assemble and place a final jar file in the configured location.

Executing `ant package` is a one-line command that will take our Java project, compile it, run unit tests, and then assemble a jar file for us. Ant is flexible, rich in functionality, and satisfies our goal of scripting a minimal build. The XML configuration file is a clean, straightforward way of documenting the project's build lifecycle. By itself, Ant is lacking in the way of dependency management. However, tools like [Apache Ivy](#) have been developed to extend this functionality to Ant.

Build With Apache Maven

According to the [Apache Maven Project documentation](#), *maven* is a Yiddish word meaning *accumulator of knowledge*. Like Apache Ant, Maven is also an open-source project of the Apache Software Foundation. It began as an improvement to the Jakarta turbine project build that was utilizing varied configurations of Ant for each subproject. It's first official release was on July, 2004.

Like Apache Ant, Maven uses an XML document, (a POM file), to describe and manage java projects. This document records information about the project including a unique identifier for the project, the required compiler version, configuration property values, and metadata on all required dependencies and their versions. One of the most powerful features of Maven is its dependency management and the ability to use repositories to share dependencies with other projects.

Maven relies heavily on convention in order to provide a uniform method of managing and documenting a project that can easily scale across all projects using Maven. A project is expected to be laid out on the filesystem in a specific way and in order to keep the script declarative, customized implementations require building custom plugins. Although it can be extensively customized to override expected defaults, Maven works out of the box with very little configuration if you conform to the expected project structure.

KEY MAVEN TERMINOLOGY

Lifecycle phase

A lifecycle phase is a discrete step in a project's build lifecycle. Maven defines a list of default phases that are executed sequentially during a build. The default phases are *validate*, *compile*, *test*, *package*, *verify*, *install*, and *deploy*. Two other Maven lifecycles consist of lifecycle phases that handle cleaning and documentation for the project. Invoking Maven with a lifecycle phase will execute all of the lifecycle phases in order up to and including the given lifecycle phase.

Maven goal

Maven goals handle the implementation details of the execution of a lifecycle phase. A goal can be configured to be associated with multiple lifecycle phases.

Maven plugin

A collection of common maven goals with a common purpose. Goals are provided by plugins to be executed in the lifecycle phase they are bound to.

POM file

The Maven *Project Object Model*, or POM, is implemented as an XML configuration file that includes the configuration for all of the Maven lifecycle phases, goals, and plugins required for the project's build lifecycle. The name of this file is *pom.xml* and is found at the root of a project. In multi-module projects, a POM file at the root of the project is potentially a *parent* POM that provides inherited configuration to POMs that specify the parent POM. All project POM files extend Maven's *Super POM* which is provided by the Maven installation itself and includes default configuration.

Example 4-2 is a simple POM file I configured for my Java 11 environment using Maven 3.6.3.

Example 4-2. Maven POM file (pom.xml)

```
<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.mycompany.app</groupId> ☐
  <artifactId>my-app</artifactId>
  <version>1.0-SNAPSHOT</version>

  <name>my-app</name>
  <!-- FIXME change it to the project's website -->
  <url>http://www.example.com</url>
```

```

<properties> ☐
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <maven.compiler.release>11</maven.compiler.release>
</properties>

<dependencies> ☐
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
    <scope>test</scope>
  </dependency>
</dependencies>

<build> ☐
  <pluginManagement>
    <plugins>
      <plugin> ☐
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.8.0</version>
      </plugin>
    </plugins>
  </pluginManagement>
</build>
</project>

```

- ☐ Every project is uniquely identified by its configured *groupId*, *artifactId*, and *_version*.
- ☐ Properties are hardcoded values that can potentially be used in multiple places in the POM file. They can be either custom properties or built-in properties used by plugins or goals.
- ☐ The *dependencies* block is where all direct dependencies of the project are identified. This project relies on junit to run the unit tests, so the junit dependency is specified here. Junit has a dependency itself on hamcrest-core, but Maven is smart enough to figure that out without having to include it here. By default, Maven will pull these dependencies from Maven Central.
- ☐ The build block is where plugins are configured. Unless there is configuration you want to override, this block isn't required.
- ☐ There are default plugin bindings for all of the lifecycle phases, but in this case, I wanted to configure the maven-compiler-plugin to use Java version 11 rather than the default. The property that controls this for the plugin is

`maven.compiler.release` in the properties block. This configuration could have been put in the plugin block, but it makes sense to move it to the properties block for better visibility toward the top of the file. This property replaces `maven.compiler.source` and `maven.compiler.target` that is usually seen when using older versions of Java.

MAVEN PLUGIN VERSIONING

It is a good idea to lock down all of your maven plugin versions to avoid using Maven defaults. Specifically, pay special attention to Maven instructions for configuring your build script when using older versions of Maven and Java versions 9 or greater. The default plugin versions of your Maven installation might not be compatible with later versions of Java.

Because of the strong reliance on convention, this Maven build script is quite brief. With this small POM file, I am able to execute `mvn package` to compile, run tests, and assemble a jar file, all utilizing default settings. If you spend any time with Apache Maven, you will quickly realize that Maven is much more than just a build tool and is chock-full of powerful features. For someone new to Maven, its potential complexity can be overwhelming. Also, customization through creating a new Maven plugin is daunting when the customization is minor. At the time of this writing, the [Apache Maven Project](#) documentation available contains excellent resources including a “Maven in 5 Minutes” guide. I highly recommend starting with these resources if you are unfamiliar with Maven.

MAVEN TO ANT CONVERSION

Although the [Apache Maven Ant Plugin](#) is no longer maintained, it is possible to generate an Ant build file from a Maven POM file. Doing this will help you appreciate everything you get out of the box with Maven’s convention and defaults! In the same directory as your *pom.xml* file, invoke the Maven plugin with the command `mvn ant:ant`

Build With Gradle

Gradle is an open-source build tool under the Apache 2.0 license. Hans Dockter, the founder of Gradle explained that his original idea was to call the project

Cradle with a **C**. He ultimately decided on the name Gradle with a **G** since it used Groovy for the DSL.⁵ Gradle 1.0 was released on June 12, 2012, so in comparison to Apache Ant and Apache Maven, Gradle is the new kid on the block.

One of the biggest differences between Gradle and Maven and Ant is that the Gradle build script is not XML based. Instead, Gradle build scripts can be written with either a Groovy or Kotlin DSL. Like Maven, Gradle also utilizes convention, but is more in the middle of the road compared to Maven. The [Gradle documentation](#) touts the flexibility of the tool and includes instructions on how to easily customize your build.

TIP

Gradle has extensive online documentation on [migrating Maven builds to Gradle](#). You can generate a Gradle build file from an existing Maven POM.

KEY GRADLE TERMINOLOGY

Domain-Specific Language (DSL)

Gradle scripts use a Domain-specific language, or DSL, specific to Gradle.⁶ With the Gradle DSL, you can write a Gradle script using either Kotlin or Groovy language features. The [Gradle Build Language Reference](#) documents the Gradle DSL.

Gradle Task

A Gradle task behaves as a discrete step in your build lifecycle. Tasks can include implementations of units of work like copying files, inputs that the implementation uses, and outputs that the implementation affects. Tasks can specify dependencies on other tasks in order to control the order of execution. Your project build will consist of a number of tasks that a Gradle build will configure and then execute in the appropriate order.

Gradle Lifecycle Tasks

These are common tasks that are provided by Gradle's Base Plugin and includes *clean*, *check*, *assemble*, and *build*. Other plugins can apply the Base Plugin for access to these tasks.

Gradle Plugin

A Gradle plugin includes a collection of Gradle tasks and is the mechanism with which to add extensions to existing functionality, features, conventions, configuration, and other customizations to your build.

Gradle Build Phase

Gradle build phases are not to be confused with Maven phases. A Gradle build will move through three fixed build phases, *initialization*, *configuration*, and *execution*.

Example 4-3 is a simple Gradle build file that I generated from the content of **Example 4-2** in the previous section.

Example 4-3. Gradle build script (build.gradle)

```
/*
 * This file was generated by the Gradle 'init' task.
 */

plugins {
    id 'java'
    id 'maven-publish'
}

repositories {
    mavenLocal()
    maven {
        url = uri('https://repo.maven.apache.org/maven2')
    }
}

dependencies {
    testImplementation 'junit:junit:4.11'
}

group = 'com.mycompany.app'
```



```

version = '1.0-SNAPSHOT'
description = 'my-app'
sourceCompatibility = '11'

publishing {
    publications {
        maven(MavenPublication) {
            from(components.java)
        }
    }
}

tasks.withType(JavaCompile) {
    options.encoding = 'UTF-8'
}

```

- ☐ Gradle plugins are applied by adding their *plugin id* to the *plugins* block. The *java* plugin is a Gradle Core plugin that provides compilation, testing, packaging, and other functionality for Java projects.
- ☐ Repositories for dependencies are provided in the *repositories* block. Dependencies are resolved using these settings.
- ☐ Gradle handles dependencies similarly to Maven. The *junit* dependency is required for our unit tests, so it is included in the *dependencies* block.
- ☐ The *sourceCompatibility* configuration setting is provided by the *java* plugin and maps to the *source* option of `javac`. There is also a *targetCompatibility* configuration setting. Its default value is the value of *sourceCompatibility*, so there was no reason to add it to the build script.
- ☐ The flexibility of Gradle allows me to add explicit encoding for the Java compiler. There is a task provided by the *java* plugin called *compileJava* that is of the type *JavaCompiler*. This code block sets the encoding property on this compile task.

This Gradle build script allows me to compile, run tests, and assemble a jar file for my project by executing the single command `gradle build`. Because Gradle builds are based on well-known conventions, build scripts only contain what is needed that differentiates the build, helping to keep them small and maintainable. This simple script shows how powerful and flexible Gradle can be, especially for Java projects that have a more complicated build process. In that case, the upfront investment required to understand the Gradle domain-specific

language for customization is well worth the time.

All three of these tools for building your Java project have their own strengths and weaknesses. You will want to choose a tool based on the needs of your project, the experience of your team, and the flexibility required. Wrangling together a build script, however you choose to do it, and with whatever tool you choose to do it, will increase your efficiency by leaps and bounds. Building a java project is a repetitive process consisting of a number of steps, ripe for human error, and marvelously suitable for automation. Reducing your project build down to a single command saves ramp-up time for new developers, increases efficiency during development tasks in a local development environment, and paves the way for build automation, an integral component of an effective CI solution. Build server options and examples will be discussed in more detail later in this chapter in the section Continuous Building in the Cloud.

Wrangling together a build script, however you choose to do it, will increase your efficiency by leaps and bounds. Building a java project is a repetitive process consisting of a number of steps, ripe for human error, and marvelously suitable for automation. Reducing your project build down to a single command saves ramp-up time for new developers, increases efficiency during development tasks in a local development environment, and paves the way for build automation, an integral component of an effective CI solution. Build server options and examples will be discussed in more detail in the section Continuous Building in the Cloud.

Automate Tests

Run Unit Tests Automatically

Monitor and Maintain Tests

Speed Up Your Test Suite

Continuously Build

-
- 1 Fowler, Martin. “Continuous Integration.” Last modified May 1, 2006.
<https://www.martinfowler.com/articles/continuousIntegration.html>
 - 2 Wells, Don. “Integrate Often.” Accessed July 5, 2020,
<http://www.extremeprogramming.org/rules/integrateoften.html>
 - 3 <http://www.extremeprogramming.org/rules/integrateoften.html>
 - 4 <http://www.extremeprogramming.org/values.html>
 - 5 Gradle forum entry in Dec, 2011 (<https://discuss.gradle.org/t/why-is-gradle-called-gradle/3226>)
 - 6 The Wikipedia entry on Domain-specific language (https://en.wikipedia.org/wiki/Domain-specific_language) provides more information.

Chapter 5. Package Management

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 6th chapter of the final book. If there is a GitHub repo associated with the book, it will be made active after final publication.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at ccollins@oreilly.com.

Somewhere in the world, as you read this sentence, a line of code is being written. This line of code will ultimately become part of an artifact that will become a building block used internally by an organization in one or more enterprise products, or shared via a public repository, most notably Maven Central for Java and Kotlin libraries.

Today, there are more libraries, binaries, and artifacts available for use than ever before, and this collection will continue to grow as developers around the world continue their next generation of products and services. Handling and managing these artifacts requires more effort now than before - with an ever-increasing number and dependencies creating a complicated web of connectedness. Using an incorrect version of an artifact is an easy trap to fall into, causing confusion, broken builds and ultimately thwarting carefully planned project release dates.

It's more important than ever for developers to not only understand the function and the idiosyncrasies of the source code directly in front of them, but to understand how their projects are packaged and how the building blocks are assembled into the final product. Having a deep understanding of the build process itself and how our automated build tools function under the hood is crucial to avert delays and hours of unnecessary troubleshooting — not to mention prevent a large category of bugs escaping into production.

Access to troves of third-party resources that provide solutions to common coding problems can help speed the development of our projects but introduce the risk of errant or unexpected behavior. Understanding how these components are brought into projects as well as a deep understanding of where they come from will help in troubleshooting efforts. Ensuring we are responsible managers of the artifacts we produce internally will allow us to improve our decision making and prioritization when it comes to bug fixes and feature development as well as help pave the way to release to production. A developer can no longer only be versed in the semantics of the code in front of them, but must also be versed in the complexities of package management.

Why build-it-and-ship-it is not enough

Not so long ago, software developers viewed building an artifact as a culmination of hard, sometimes epic, efforts. Meeting deadlines sometimes meant using shortcuts and poorly documented steps. Since then, the requirements of the industry have changed to bring faster delivery cycles, diverse environments, tailored artifacts, exploding codebases and repositories, and multi-module packages. Today building an artifact is just one step of a bigger business cycle.

Successful leaders recognize that the best innovations emerge out of trial and error. That's why they've made testing, experimentation and failure an integral part of their lives and their company's process.

One way to innovate, scale more quickly, launch more products, improve the quality or user experience of applications or products, and roll out new features is through A/B testing.

You may be asking, what is A/B testing? According to Kaiser Fung, who founded the applied analytics program at Columbia University, A/B testing, at its most basic, is a century old method used to compare two versions of something to figure out which performs better. Today several startups, well established companies like Microsoft, and several other leading companies — including Amazon, Booking.com, Facebook, and Google — each have been conducting more than 10,000 online controlled experiments annually.¹

Booking.com conducts comparative testing on every new feature on its website

BOOKING.COM conducts comparative testing on every new feature on its website, comparing details from the selection of photos and content to button color and placement. By testing several versions against each other and tracking customer response, the company is able to constantly improve their user experience.

How do we deliver and deploy multiple versions of software composed of multiple different artifacts? How do we find bottle-necks? How do we know we are moving in the right direction? How do we keep track of what is working well or what is working against us? How do we maintain a reproducible outcome but with enriched lineages? Answers to these questions can be found by capturing and analysing relevant, contextual, clear and specific information regarding the workflows and artifacts' inputs, outputs and states. All of this is possible thanks to metadata.

It's all about metadata

In God we trust, all others bring data.

—W Edwards Deming

What's metadata?

Metadata is defined as a structured key/value storage of relevant information, or in other words, a collection of properties or attributes applicable to a particular entity, which in our case applies to artifacts and processes.

Metadata enables the discovery of correlations and causations as well as insights into the organization's behaviour and outcomes. As a result this will show if the organization is tuned in to their stakeholders goals.

Data allows different narratives to have a more complete overview. It is important to have strong and solid building blocks. A good starting point is to answer the following questions concerning the main stages of each phase of the software development cycle: Whom? What? How? Where? and When? Asking the right questions is only half of the effort, having clear, relevant, specific and clear answers that can be normalized or enumerated is always a good practice.

Contextualized

All data needs to be interpreted within a frame of reference. In order to

visibility

Not all consumers are interested in all data.

Format and encoding

One specific property may be exposed during different stages in different formats, but there needs to be consistency in the naming, meaning and possibly the general value.

Let's turn our attention to generating and packaging metadata with build tools. The Java ecosystem has no shortage of options when it comes to build tools, arguably the most popular are Apache Maven and Gradle thus it makes sense to discuss them in depth. However should your build depend on a different build tool it's likely that the information presented in this section will still prove to be of use, as some of the techniques to gather and package metadata may be reused.

Now, before we jump into practical code snippets there are three action items that we have to figure out. First, determine the metadata that should be packaged with an artifact. Second, find out how said metadata can be obtained during the build. Third, process the metadata and record it in the appropriate format or formats. The following sections cover each one of these aspects.

Determining the metadata

The build environment has no shortage of information that can be converted into metadata and packaged alongside an artifact. A good example is the build timestamp that identifies the time and date when the build produced the artifact. There are many timestamp formats that can be followed but I'd recommend using **ISO 8601** whose formatted representation using `java.text.SimpleDateFormat` is `yyyy-MM-dd'T'HH:mm:ssXXX` — useful when the captured timestamp relies on `java.util.Date`.

Alternatively the

`java.time.format.DateTimeFormatter.ISO_OFFSET_DATE_TIME` may be used if the captured timestamp relies on `java.time.LocalDateTime`. The build's OS details may also be of interest, as well as JDK information such as version, id, vendor. Luckily for us these bits of information are captured by the JVM and exposed via **System**

properties.

Consider including the artifact's id and version as well, (even though these values are usually encoded in the artifact's filename), as a precaution in case the artifact were to be renamed at some point. Source Code Management (SCM) information is also crucial. Useful information from source control includes the commit hash, tag, and the branch name. Additionally you may want to capture specific build information such as the user that runs the build; the build tool's name, id, and version; the hostname and IP address of the build machine. These key/value pairs are likely the most significant and commonly found metadata however you may select additional key/value pairs that are required by other tools and systems that will consume the produced artifacts.

Of course I can't stress enough how important it is to check your team's and organization's policies regarding access and visibility of sensitive data. Some of the key/value pairs mentioned before may be deemed a security risk if exposed to third parties or external consumers, however they may be of high importance to internal consumers.

Capturing metadata

We must find a way to gather said metadata with our build tool of choice after we have determined which metadata we need to capture. Some of the key/value pairs can be obtained directly from the environment, system settings, and command flags exposed by the JVM as environment variables or `System` properties. Additional properties may be exposed by the build tool itself, whether they may be defined as additional command line arguments or as configuration elements in the tool's configuration settings.

Let's assume for the moment that we need to capture the following key/value pairs:

- JDK information such as version and vendor.
- OS information such as name, arch, and version.
- The build timestamp.
- The current commit hash from SCM (assuming Git).

These values may be captured with Apache Maven using a combination of `System` properties for the first two items and a third party plugin for the last two. Both Apache Maven and Gradle have no shortage of options when it comes to plugins that offer integration with Git however I'd recommend choosing `git-commit-id-maven-plugin` for Apache Maven and `versioning` for Gradle as these plugins are the most versatile so far.

Now, Apache Maven allows defining properties in a handful of ways, most commonly as key/value pairs inside the `<properties>` section found in the `pom.xml` build file. The value for each key is free text although you can refer to `System` properties using a shorthand notation or to environment variables using a naming convention. Say you want to access the value for the `"java.version"` key found in `System` properties. This can be done by referring to it using the `${}` shorthand notation such as `${java.version}`. Conversely for an environment variable you may use the `${env.<NAME>}` notation. For example, the value of an environment variable named `TOKEN` can be accessed using the expression `${env.TOKEN}` in the `pom.xml` build file. Putting together the `git-commit-id` plugin and build properties may result in a `pom.xml` similar to the following.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.acme</groupId>
  <artifactId>example</artifactId>
  <version>1.0.0-SNAPSHOT</version>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <build.jdk>${java.version} (${java.vendor} ${java.vm.version})
  </build.jdk>
    <build.os>${os.name} ${os.arch} ${os.version}</build.os>
    <build.revision>${git.commit.id}</build.revision>
    <build.timestamp>${git.build.time}</build.timestamp>
  </properties>

  <build>
    <plugins>
      <plugin>
```

```

<groupId>pl.project13.maven</groupId>
<artifactId>git-commit-id-plugin</artifactId>
<version>4.0.3</version>
<executions>
  <execution>
    <id>resolve-git-properties</id>
    <goals>
      <goal>revision</goal>
    </goals>
  </execution>
</executions>
<configuration>
  <verbose>>false</verbose>
  <failOnNoGitDirectory>>false</failOnNoGitDirectory>
  <generateGitPropertiesFile>>true</generateGitPropertiesFile>

  <generateGitPropertiesFilename>${project.build.directory}/git.properties
</generateGitPropertiesFilename>
  <dateFormat>yyyy-MM-dd'T'HH:mm:ssXXX</dateFormat>
</configuration>
</plugin>
</plugins>
</build>
</project>

```

Note that the values for `build.jdk` and `build.os` already include formatting as they are composites of simpler values whereas the `build.revision` and `build.timestamp` values come from the properties defined by the git plugin. We have yet to determine the final format and file or files that will contain the metadata, which is why we see it defined in the `<properties>` section. This setup allows these values to be reused and consumed by other plugins should they need it. Another reason to prefer this setup is that external tools (such as those found in a build pipeline) may read these values more easily as they are located at a specific section instead of at many places within the build file.

Also note the chosen value of version, it's `1.0.0-SNAPSHOT`. You may use any character combination for the version as you deem necessary however it's customary to at least use an alphanumeric sequence that defines two numbers in the `<major>.<minor>` format. There are a handful of versioning conventions out there with both advantages and drawbacks. This being said the use of the `-SNAPSHOT` tag has a special meaning as it indicates the artifact is not yet ready

for production. Some tools will behave differently when a snapshot version is detected, for example they can prevent an artifact from ever being published to a production environment.

Opposed to Apache Maven, Gradle has no shortage of options when it comes to defining and writing builds files. To begin with, as of Gradle 6.8, you have two choices for the build file format: Apache Groovy DSL or Kotlin DSL.

Regardless of which one you pick you will soon find that there are more options to capture and format metadata. Some of them may be very idiomatic, some may require additional plugins, some may even be considered outdated or obsolete. To keep things short and basic we'll go with the Apache Groovy DSL and small idiomatic expressions. We'll capture the same metadata similarly as we did for Apache Maven, with the first two values coming from `System` properties and the commit hash provided by the `versioning` git plugin, but the build timestamp will be calculated on the spot using custom code. The following snippet shows how this can be done.

```
plugins {
    id 'java-library'
    id 'net.nemerosa.versioning' version '2.14.0'
}

version = '1.0.0-SNAPSHOT'

ext {
    buildJdk = [
        System.properties['java.version'],
        '(' + System.properties['java.vendor'],
        System.properties['java.vm.version'] + ')'
    ].join(' ')
    buildOs = [
        System.properties['os.name'],
        System.properties['os.arch'],
        System.properties['os.version']
    ].join(' ')
    buildRevision = project.extensions.versioning.info.commit
    buildTimestamp = new Date().format("yyyy-MM-dd'T'HH:mm:ssXXX")
}
```

These computed values will be available as dynamic project properties which may be consumed later in the build by additional configured elements such as extensions, tasks, closures (for Apache Groovy), actions (for Apache Groovy and Kotlin) and other elements supported by the DSL. All that is left to do is

and `KoumIn`), and other elements exposed by the DSL. All that is left now is recording the metadata in a given format, which we'll cover next.

Writing the metadata

You may find that you must record metadata in more than one format or files. The choice of format depends on the intended consumers. Some consumers require a unique format that no other consumer can read while others may understand a variety of formats. Be sure to consult the documentation of a given consumer on its supported formats and options and also check if integration with your build tool of choice is provided. It's possible that you discover that a plugin for your build is available that eases the recording process of the metadata that you need. For demonstration purposes we'll record the metadata using two popular formats: a Java properties file and the JAR's manifest.

We can leverage Apache Maven's **resource filtering** which is baked into the **resources plugin**, part of the core set of plugins that every build has access to. For this to work we must add the following snippet to the previous `pom.xml` file, inside the `<build>` section.

```
<resources>
  <resource>
    <directory>src/main/resources</directory>
    <filtering>true</filtering>
  </resource>
</resources>
```

A companion properties file located at `src/main/resources` is also required. I've chosen `META-INF/metadata.properties` as the relative path and name of the properties file to be found inside the artifact JAR. Of course you may choose a different naming convention as needed. This file relies on variable placeholder substitutions, variables which will be resolved from project properties such as those we set in the `<properties>` section. By following convention, there is little configuration information needed in the build file. The properties file looks like the following:

```
build.jdk      = ${build.jdk}
build.os       = ${build.os}
build.revision = ${build.revision}
```

```
build.timestamp = ${build.timestamp}
```

Recording the metadata in the JAR's manifest requires tweaking the configuration of the `jar-maven-plugin` applicable to a build file. The following snippet must be included inside the `<plugins>` section found in the `<build>` section. In other words, it's a sibling of the `git-commit-id` plugin we saw earlier in this section.

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-jar-plugin</artifactId>
  <version>3.2.0</version>
  <configuration>
    <archive>
      <manifestEntries>
        <Build-Jdk>${build.jdk}</Build-Jdk>
        <Build-OS>${build.os}</Build-OS>
        <Build-Revision>${build.revision}</Build-Revision>
        <Build-Timestamp>${build.timestamp}</Build-Timestamp>
      </manifestEntries>
    </archive>
  </configuration>
</plugin>
```

Note that a specific plugin version is defined even though this plugin is part of the core plugin set. The reason behind this is that it's imperative to declare all plugin versions for the sake of reproducible builds, otherwise you will find builds may differ as different plugin versions may be resolved depending on the specific version of Apache Maven used to run the build. Each entry in the manifest is composed of a capitalized key and the captured value. Running the build with `mvn package` resolves the captured properties, copies the metadata properties file with resolved values into the `target/classes` directory where it will be added to the final JAR, and injects the metadata into the JAR's manifest. We can verify this by inspecting the contents of the generated artifact.

```
$ mvn verify
$ jar tvf target/example-1.0.0-SNAPSHOT.jar
 0 Sun Jan 10 20:41:12 CET 2021 META-INF/
131 Sun Jan 10 20:41:10 CET 2021 META-INF/MANIFEST.MF
205 Sun Jan 10 20:41:12 CET 2021 META-INF/metadata.properties
 0 Sun Jan 10 20:41:12 CET 2021 META-INF/maven/
 0 Sun Jan 10 20:41:12 CET 2021 META-INF/maven/com.acme/
 0 Sun Jan 10 20:41:12 CET 2021 META-INF/maven/com.acme/example/
```

```
0 Sun Jan 10 20:41:12 CET 2021 META-INF/maven/com.acme/example/
1693 Sun Jan 10 19:13:52 CET 2021 META-
INF/maven/com.acme/example/pom.xml
109 Sun Jan 10 20:41:12 CET 2021 META-
INF/maven/com.acme/example/pom.properties
```

The two files are found inside the JAR file as expected. Extracting the JAR and looking at the contents of the properties file and the JAR manifest yield the following results

```
build.jdk      = 11.0.9 (Azul Systems, Inc. 11.0.9+11-LTS)
build.os       = Mac OS X x86_64 10.15.7
build.revision = 0ab9d51a3aaa17fca374d28be1e3f144801daa3b
build.timestamp = 2021-01-10T20:41:11+01:00
```

```
Manifest-Version: 1.0
Created-By: Maven Jar Plugin 3.2.0
Build-Jdk-Spec: 11
Build-Jdk: 11.0.9 (Azul Systems, Inc. 11.0.9+11-LTS)
Build-OS: Mac OS X x86_64 10.15.7
Build-Revision: 0ab9d51a3aaa17fca374d28be1e3f144801daa3b
Build-Timestamp: 2021-01-10T20:41:11+01:00
```

We've covered how to collect metadata with Apache Maven. Let's show the same method of recording metadata using a properties file and JAR manifest using Gradle. For the first part we'll configure the standard `processResources` task that's provided by the `java-library` plugin we applied to the build. The additional configuration can be appended to the previously shown Gradle build file, and it looks like the following:

```
processResources {
    expand(
        'build_jdk'      : project.buildJdk,
        'build_os'       : project.buildOs,
        'build_revision' : project.buildRevision,
        'build_timestamp' : project.buildTimestamp
    )
}
```

Take note that the names of the keys use `_` as a token separator, this is due to the default resource filtering mechanism used by Gradle. If we were to use `.` as we saw earlier with Apache Maven, then Gradle would expect to find a `build`

“object” with matching “jdk”, “os”, “revision”, and “timestamp” properties during resource filtering. That object does not exist which will cause the build to fail. Changing the token separator avoids that problem but also forces us to change the contents of the properties file to the following:

```
build.jdk      = ${build_jdk}
build.os       = ${build_os}
build.revision = ${build_revision}
build.timestamp = ${build_timestamp}
```

Configuring the JAR manifest is a straight forward operation given that the `jar` task offers an entry point for this behavior, as shown by the following snippet that can also be appended to the existing Gradle build file.

```
jar {
    manifest {
        attributes(
            'Build-Jdk'       : project.buildJdk,
            'Build-OS'        : project.buildOs,
            'Build-Revision'  : project.buildRevision,
            'Build-Timestamp' : project.buildTimestamp
        )
    }
}
```

As seen before, each manifest entry uses a capitalized key and its corresponding captured value. Running the build with `gradle jar` should produce results similar to those provided by Apache Maven, that is, the properties file will be copied to a target location where it can be included in the final JAR, with its value placeholders substituted for the actual metadata values, and the JAR manifest will be enriched with metadata as well. Inspecting the JAR shows that it contains the expected files.

```
$ gradle jar
$ jar tvf build/libs/example-1.0.0-SNAPSHOT.jar
 0 Sun Jan 10 21:08:22 CET 2021 META-INF/
25 Sun Jan 10 21:08:22 CET 2021 META-INF/MANIFEST.MF
165 Sun Jan 10 21:08:22 CET 2021 META-INF/metadata.properties
```

Unpacking the JAR and looking inside each file yields the following results:

```
build.jdk      = 11.0.0 (Adopt Systems, Inc., 11.0.0.11 LTS)
```



```
build.jdk      = 11.0.9 (Azul Systems, Inc. 11.0.9+11-LTS)
build.os       = Mac OS X x86_64 10.15.7
build.revision = 0ab9d51a3aaa17fca374d28be1e3f144801daa3b
build.timestamp = 2021-01-10T21:08:22+01:00
```

```
Manifest-Version: 1.0
Build-Jdk: 11.0.9 (Azul Systems, Inc. 11.0.9+11-LTS)
Build-OS: Mac OS X x86_64 10.15.7
Build-Revision: 0ab9d51a3aaa17fca374d28be1e3f144801daa3b
Build-Timestamp: 2021-01-10T21:08:22+01:00
```

Perfect! That is all that there is to it. Let me encourage you to add or remove key/values pairs as needed as well as configure other plugins (for both Apache Maven and Gradle) that may expose additional metadata or provide other means to process and record metadata into particular formats.

Dependency management basics for Apache Maven & Gradle

Dependency management has been a staple of Java projects since Apache Maven 1.x came to light in 2002. The gist behind this feature is to declare artifacts that are required for compiling, testing, and consuming a particular project, relying on additional metadata attached to an artifact such as its group identifier, artifact identifier, version, and sometimes a classifier as well. This metadata is typically exposed using a well known file format: the **Apache Maven POM** expressed in a `pom.xml` file. Other build tools are capable of understanding this format, and can even produce and publish `pom.xml` files despite using a totally unrelated format for declaring build aspects, as is the case for Gradle with `build.gradle` (Apache Groovy DSL) or `build.gradle.kts` (Kotlin DSL) build file.

Despite being a core feature provided by Apache Maven since the early days, and also a core feature in Gradle, dependency management and dependency resolution remains a stumbling block for many, as even though the rules to declare dependencies are not complicated, you may find yourself at the mercy of published metadata with invalid, misleading, or missing constraints. The following sections are a primer for dependency management using both Apache Maven and Gradle but it is by no means an exhaustive explanation — that would take a whole book! In other words, tread carefully dear reader, there be dragons

take a whole book: in other words, read carefully dear reader, there be dragons ahead. I'll do my best to point out the safest paths.

We'll begin with Apache Maven as it is the build tool that defines the artifact metadata using the `pom.xml` file format.

Dependency management with Apache Maven

It's very likely you have encountered a POM file before, after all it's quite ubiquitous. A POM file with model version `4.0.0` is responsible for defining how artifacts are produced and consumed. In Apache Maven version 4 these two capabilities are split although the model version remains the same for compatibility reasons. It's expected that the model format will change when Apache Maven version `5.0.0` is introduced however there are no details on how this model will look at the time of writing this book. One thing is sure, the Apache Maven developers are very keen in keeping backward compatibility. Let's walk through the basics, shall we?

Dependencies are identified by 3 required elements: `groupId`, `artifactId`, and `version`. These elements are collectively known as “Maven coordinates”, or GAV coordinates, where GAV, as you may have guessed, stands for “`groupId`, `artifactId`, `version`”. From time to time you may find dependencies that define a fourth element named `classifier`. Let's break them down one by one. Both `artifactId` and `version` are straightforward, the former defines the “name” of the artifact while the latter defines a version number. There may be many different versions that are associated with the same `artifactId`. The `groupId` is used to put together a set of artifacts that have some sort of relationship, that is, all of them belong to the same project or provide behavior that's germane to one another. The `classifier` adds another dimension to the artifact, albeit optional. Classifiers are often used to differentiate artifacts that are specific to a particular setting such as the operating system or the Java release. Examples of operating system classifiers are found in the JavaFX binaries, such as `javafx-controls-15-win.jar`, `javafx-controls-15-mac.jar`, and `javafx-controls-15-linux.jar` which identify version 15 of the JavaFX controls binaries that may be used with Windows, Mac OS, and Linux platforms. Another set of common classifiers are `sources` and `javadoc` which identify JAR files that contain sources and generated

documentation (via javadoc). The combination of GAV coordinates must be unique otherwise the dependency resolution mechanism will have a hard time finding out correct dependencies to use.

POM files let you define dependencies inside the `<dependencies>` section, where you would list GAV coordinates for each dependency. In its most simple for it looks something like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd"
  xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.acme</groupId>
  <artifactId>example</artifactId>
  <version>1.0.0-SNAPSHOT</version>

  <dependencies>
    <dependency>
      <groupId>org.apache.commons</groupId>
      <artifactId>commons-collections4</artifactId>
      <version>4.4</version>
    </dependency>
  </dependencies>
</project>
```

Dependencies listed in this way are known as “direct” dependencies as they are explicitly declared in the POM file. This classification holds true even for dependencies that may be declared in a POM that’s marked as a parent of the current POM. What’s a parent POM? It’s just like another pom.xml file except that your POM marks it with a parent/child relationship using the `<parent>` section. In this way configuration defined by the parent POM can be inherited by the child POM. We can inspect the dependency graph by invoking the `mvn dependency:tree` command, which resolves the dependency graph and prints it out.

```
$ mvn dependency:tree
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.acme:example >-----
-----
[INFO] Building example 1.0.0-SNAPSHOT
-----
```

```

[INFO] -----[ jar ]-----
[INFO]
[INFO] --- maven-dependency-plugin:2.8:tree (default-cli) @ example ---
[INFO] com.acme:example:jar:1.0.0-SNAPSHOT
[INFO] \- org.apache.commons:commons-collections4:jar:4.4:compile
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
-----

```

Here we can see that the current POM (identified by its GAV coordinates as `com.acme:example:1.0.0-SNAPSHOT`) has a single, direct dependency. There are two additional elements found in the output of the `commons-collections4` dependency: the first being `jar` which identifies the type of the artifact, the second being `compile` which identifies the scope of the dependency. We'll come back to scopes in a moment, but suffice to say that if there is no explicit `<scope>` element defined for a dependency, then its default scope becomes `compile`. Now, when a POM that contains direct dependencies is consumed it brings along those dependencies as transitive from the point of view of the consuming POM. The next example shows that particular setup.

```

<?xml version="1.0" encoding="UTF-8"?>
<project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd"
  xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.acme</groupId>
  <artifactId>example</artifactId>
  <version>1.0.0-SNAPSHOT</version>

  <dependencies>
    <dependency>
      <groupId>commons-beanutils</groupId>
      <artifactId>commons-beanutils</artifactId>
      <version>1.9.4</version>
    </dependency>
  </dependencies>
</project>

```

Resolving and printing out the dependency graph using the same command as

before yields this result:

```
$ mvn dependency:tree
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.acme:example >-----
-----
[INFO] Building example 1.0.0-SNAPSHOT
[INFO] -----[ jar ]-----
-----
[INFO]
[INFO] --- maven-dependency-plugin:2.8:tree (default-cli) @ example ---
-
[INFO] com.acme:example:jar:1.0.0-SNAPSHOT
[INFO] \- commons-beanutils:commons-beanutils:jar:1.9.4:compile
[INFO]    +- commons-logging:commons-logging:jar:1.2:compile
[INFO]    \- commons-collections:commons-collections:jar:3.2.2:compile
[INFO] -----
-----
[INFO] BUILD SUCCESS
[INFO] -----
-----
```

This tells us that the `commons-beanutils` artifact has 2 dependencies set in the `compile` scope which from the point of view of the `com.acme:example:1.0.0-SNAPSHOT` happen to be seen as transitive. It appears that these 2 transitive dependencies have no direct dependencies of their own as there's nothing listed for either of them, however if you were to look at the `commons-logging` POM file you'll find the following dependency declarations:

```
<dependencies>
  <dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.17</version>
    <optional>true</optional>
  </dependency>
  <dependency>
    <groupId>logkit</groupId>
    <artifactId>logkit</artifactId>
    <version>1.0.1</version>
    <optional>true</optional>
  </dependency>
</dependencies>
```

```

    <groupId>avalon-framework</groupId>
    <artifactId>avalon-framework</artifactId>
    <version>4.1.5</version>
    <optional>true</optional>
</dependency>
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>servlet-api</artifactId>
    <version>2.3</version>
    <scope>provided</scope>
    <optional>true</optional>
</dependency>
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
    <scope>test</scope>
</dependency>
</dependencies>

```

As you can see there are actually 5 dependencies! However 4 of them define an additional `<optional>` element while 2 of them define a different value for `<scope>`. Dependencies marked as `<optional>` may be required for compiling and testing the producer (commons - logging in this case) but not necessarily for consumers. It depends on a case by case basis. It's time to discuss scopes now that we see them once again.

Scopes determine if a dependency is to be included in the classpath or not, as well as limiting their transitivity. Apache Maven defines 6 scopes as follows:

- **compile:** The default scope, used if none is specified as we saw earlier. Dependencies in this scope will be used for all classpaths in the project (compile, runtime, test) and will be propagated to consuming projects.
- **provided:** Like **compile** except that it does not affect the runtime classpath nor it is transitive. Dependencies set in this scope are expected to be provided by the hosting environment, such as it is the case for web applications packaged as WARs and launched from within an application server.
- **runtime:** This scope indicates the dependency is not required for compilation but for it is for execution. Both the runtime and test classpaths include dependencies set in this scope, while the compile

classpath is ignored.

- **test:** Defines dependencies required for compiling and running tests. This scope is not transitive.
- **system:** Similar to `provided` except that dependencies must be listed with an explicit path, relative or absolute. It's because of this reason that this scope is seen as a bad practice and should be avoided at all costs. There are however a handful of use cases when it may come in handy but you must bear the consequences. At best it's an option left to the experts — in other words imagine that this scope does not exist at all.
- **import:** This scope applies only to dependencies of type `pom` (the default is `jar` if not specified) and can only be used for dependencies declared inside the `<dependencyManagement>` section. Dependencies in this scope are replaced by the list of dependencies found in their own `<dependencyManagement>` section.

The `<dependencyManagement>` section has three purposes: the first is to provide version hints for transitive dependencies, the second is to provide a list of dependencies that may be imported using the `import` scope, and the third is to provide a set of defaults when used in a parent-child POM combination. Let's have a look at the first purpose. Say you have the the following dependencies defined in your POM file:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd"
  xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.acme</groupId>
  <artifactId>example</artifactId>
  <version>1.0.0-SNAPSHOT</version>

  <dependencies>
    <dependency>
      <groupId>com.google.inject</groupId>
      <artifactId>guice</artifactId>
      <version>4.2.2</version>
    </dependency>
  </dependencies>
```

```

    <groupId>com.google.truth</groupId>
    <artifactId>truth</artifactId>
    <version>1.0</version>
  </dependency>
</dependencies>
</project>

```

Both the guice and truth artifacts define guava as a direct dependency. This means guava is seen as a transitive dependency from the consumer's point of view. We get the following result if we resolve and print out the dependency graph:

```

$ mvn dependency:tree
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.acme:example >-----
-----
[INFO] Building example 1.0.0-SNAPSHOT
[INFO] -----[ jar ]-----
-----
[INFO]
[INFO] --- maven-dependency-plugin:2.8:tree (default-cli) @ example ---
-
[INFO] com.acme:example:jar:1.0.0-SNAPSHOT
[INFO] +- com.google.inject:guice:jar:4.2.2:compile
[INFO] | +- javax.inject:javax.inject:jar:1:compile
[INFO] | +- aopalliance:aopalliance:jar:1.0:compile
[INFO] | \- com.google.guava:guava:jar:25.1-android:compile
[INFO] |     +- com.google.code.findbugs:jsr305:jar:3.0.2:compile
[INFO] |     +- com.google.j2objc:j2objc-annotations:jar:1.1:compile
[INFO] |     \- org.codehaus.mojo:animal-sniffer-
[INFO] annotations:jar:1.14:compile
[INFO] \- com.google.truth:truth:jar:1.0:compile
[INFO]     +- org.checkerframework:checker-compat-
[INFO] qual:jar:2.5.5:compile
[INFO]     +- junit:junit:jar:4.12:compile
[INFO]     | \- org.hamcrest:hamcrest-core:jar:1.3:compile
[INFO]     +- com.googlecode.java-diff-
[INFO] utils:diffutils:jar:1.3.0:compile
[INFO]     +- com.google.auto.value:auto-value-
[INFO] annotations:jar:1.6.3:compile
[INFO]     \-
[INFO] com.google.errorprone:error_prone_annotations:jar:2.3.1:compile
[INFO] -----
-----
[INFO] BUILD SUCCESS
[INFO] -----
-----

```


The resolved version of guava turns out to be 25.1-android because that's the version found first in the graph. Look what happens if we invert the order of the dependencies and list truth before guice and resolve the graph once again.

```
$ mvn dependency:tree
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.acme:example >-----
-----
[INFO] Building example 1.0.0-SNAPSHOT
[INFO] -----[ jar ]-----
-----
[INFO]
[INFO] --- maven-dependency-plugin:2.8:tree (default-cli) @ example ---
-
[INFO] com.acme:example:jar:1.0.0-SNAPSHOT
[INFO] +- com.google.truth:truth:jar:1.0:compile
[INFO] | +- com.google.guava:guava:jar:27.0.1-android:compile
[INFO] | | +- com.google.guava:failureaccess:jar:1.0.1:compile
[INFO] | | +- com.google.guava:listenablefuture:jar:9999.0-empty-to-
avoid-conflict-with-guava:compile
[INFO] | | +- com.google.code.findbugs:jsr305:jar:3.0.2:compile
[INFO] | | +- com.google.j2objc:j2objc-annotations:jar:1.1:compile
[INFO] | | \- org.codehaus.mojo:animal-sniffer-
annotations:jar:1.17:compile
[INFO] | +- org.checkerframework:checker-compat-
qual:jar:2.5.5:compile
[INFO] | +- junit:junit:jar:4.12:compile
[INFO] | | \- org.hamcrest:hamcrest-core:jar:1.3:compile
[INFO] | +- com.googlecode.java-diff-
utils:diffutils:jar:1.3.0:compile
[INFO] | +- com.google.auto.value:auto-value-
annotations:jar:1.6.3:compile
[INFO] | \-
com.google.errorprone:error_prone_annotations:jar:2.3.1:compile
[INFO] \- com.google.inject:guice:jar:4.2.2:compile
[INFO]     +- javax.inject:javax.inject:jar:1:compile
[INFO]     \- aopalliance:aopalliance:jar:1.0:compile
[INFO] -----
-----
[INFO] BUILD SUCCESS
[INFO] -----
-----
```

Now the resolved version of guava happens to be 27.0.1-android because

it's the one found first in the graph. This particular behavior is a constant source of head scratching moments and frustration. As developers we are used to versioning conventions, most notably what's known as **semantic versioning** when it applies to dependency versions. Semantic versioning tells us that version tokens (separated by . dots) have specific meaning based on their position. The first token identifies the major release, the second token identifies the minor release, and the third token identifies the build/patch/fix/revision release. It's also customary that version 27.0.1 is seen as more recent than 25.1.0 because the major number 27 is greater than 25. In our case we have two different versions for guava in the graph, 27.0.1-android and 25.1.0-android, and both of them are found at the same distance from the current POM, that is, just one level down in the transitive graph. It's easy to assume that because we as developers are aware of semantic versioning and can clearly determine which version is more recent, so can Apache Maven — and that is where assumption clashes with reality! As a matter of fact Apache Maven never looks at the version, it only looks at the location within the graph, this is why we get different results if we change the order of dependencies. We can use the `<dependencyManagement>` section to fix this issue.

Dependencies defined in the `<dependencyManagement>` section usually have the 3 main GAV coordinates. When Apache Maven resolves dependencies it will look at the definitions found in this section to see if there's a match for `groupId` and `artifactId`, in which case the associated `version` will be used. It does not matter how deep in the graph a dependency may be, or how many times it may be found in the graph, if there's a match then that explicit version will be the chosen one. We can verify this claim by adding a `<dependencyManagement>` section to the consumer POM that looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd"
  xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.acme</groupId>
  <artifactId>example</artifactId>
  <version>1.0.0-SNAPSHOT</version>
```

```

<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>com.google.guava</groupId>
      <artifactId>guava</artifactId>
      <version>29.0-jre</version>
    </dependency>
  </dependencies>
</dependencyManagement>

<dependencies>
  <dependency>
    <groupId>com.google.truth</groupId>
    <artifactId>truth</artifactId>
    <version>1.0</version>
  </dependency>
  <dependency>
    <groupId>com.google.inject</groupId>
    <artifactId>guice</artifactId>
    <version>4.2.2</version>
  </dependency>
</dependencies>
</project>

```

We can see that the declaration for guava uses the `com.google.guava:guava:29.0-jre` coordinates, meaning that version `29.0-jre` will be used if there happens to be a transitive dependency that matches the given `groupId` and `artifactId`, which we know will happen in our consumer POM, twice to be exact. We get the following result when resolving and printing out the dependency graph:

```

$ mvn dependency:tree
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.acme:example >-----
-----
[INFO] Building example 1.0.0-SNAPSHOT
[INFO] -----[ jar ]-----
-----
[INFO]
[INFO] --- maven-dependency-plugin:2.8:tree (default-cli) @ example ---
[INFO] com.acme:example:jar:1.0.0-SNAPSHOT
[INFO] +- com.google.truth:truth:jar:1.0:compile
[INFO] | +- com.google.guava:guava:jar:29.0-jre:compile
[INFO] | | +- com.google.guava:failureaccess:jar:1.0.1:compile
[INFO] | | +- com.google.guava:listenablefuture:jar:9999.0-empty-to-

```

```

avoid-conflict-with-guava:compile
[INFO] | | +- com.google.code.findbugs:jsr305:jar:3.0.2:compile
[INFO] | | +- org.checkerframework:checker-qual:jar:2.11.1:compile
[INFO] | | \- com.google.j2objc:j2objc-annotations:jar:1.3:compile
[INFO] | +- org.checkerframework:checker-compat-
qual:jar:2.5.5:compile
[INFO] | +- junit:junit:jar:4.12:compile
[INFO] | | \- org.hamcrest:hamcrest-core:jar:1.3:compile
[INFO] | +- com.googlecode.java-diff-
utils:diffutils:jar:1.3.0:compile
[INFO] | +- com.google.auto.value:auto-value-
annotations:jar:1.6.3:compile
[INFO] | \-
com.google.errorprone:error_prone_annotations:jar:2.3.1:compile
[INFO] \- com.google.inject:guice:jar:4.2.2:compile
[INFO]     +- javax.inject:javax.inject:jar:1:compile
[INFO]     \- aopalliance:aopalliance:jar:1.0:compile
[INFO] -----
-----
[INFO] BUILD SUCCESS
[INFO] -----
-----

```

Note that the chosen version for guava is indeed 29.0-jre and not the previous versions we saw earlier in this chapter, confirming that the `<dependencyManagement>` section is performing its job as expected.

The second purpose of the `<dependencyManagement>`, listing dependencies that may be imported, is accomplished by using the `import` scope alongside dependencies of type `pom`. These types of dependencies usually define `<dependencyManagement>` sections of their own although nothing stops these POMs from adding more sections. POM Dependencies that define a `<dependencyManagement>` section and no `<dependencies>` section are known as Bill of Materials or BOM for short. Typically BOM dependencies define a set of artifacts that belong together for a particular purpose. Although not explicitly defined in the Apache Maven documentation you can find two kinds of BOM dependencies:

- **library:** All declared dependencies belong to the same project even though they might have different groupIds, and possibly even different versions. An example can be found at [helidon-bom](#) which groups all artifacts from the Helidon project.

- **stack:** Dependencies are grouped by behavior and the synergy they bring. Dependencies may belong to disparate projects. An example can be found at [helidon-dependencies](#) which groups the previous helidon-bom with other dependencies such as Netty, logging, etc.

Let's take the helidon-dependencies as a source of dependencies. Inspecting this POM we find dozens of dependencies declared inside its `<dependencyManagement>` section, only a few of which are seen in the following snippet:

```
<artifactId>helidon-dependencies</artifactId>
<packaging>pom</packaging>
<!-- additional elements ellided -->
<dependencyManagement>
  <dependencies>
    <!-- more dependencies ellided -->
    <dependency>
      <groupId>io.netty</groupId>
      <artifactId>netty-handler</artifactId>
      <version>4.1.51.Final</version>
    </dependency>
    <dependency>
      <groupId>io.netty</groupId>
      <artifactId>netty-handler-proxy</artifactId>
      <version>4.1.51.Final</version>
    </dependency>
    <dependency>
      <groupId>io.netty</groupId>
      <artifactId>netty-codec-http</artifactId>
      <version>4.1.51.Final</version>
    </dependency>
    <!-- more dependencies ellided -->
  </dependencies>
</dependencyManagement>
```

Consuming this BOM dependency in our own POM requires the use of the `<dependencyManagement>` section once again. We'll also define an explicit dependency for `netty-handler` like we have done before when defining dependencies except that this time we'll omit the `<version>` element. The POM ends up looking like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
```

```

http://maven.apache.org/xsd/maven-4.0.0.xsd"
  xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<modelVersion>4.0.0</modelVersion>
<groupId>com.acme</groupId>
<artifactId>example</artifactId>
<version>1.0.0-SNAPSHOT</version>

<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>io.helidon</groupId>
      <artifactId>helidon-dependencies</artifactId>
      <version>2.2.0</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>

<dependencies>
  <dependency>
    <groupId>io.netty</groupId>
    <artifactId>netty-handler</artifactId>
  </dependency>
</dependencies>
</project>

```

Please note how the `helidon-dependencies` dependency was imported. There is a key element that must be defined, `<type>`, which must be set as `pom`. Remember earlier in this chapter that dependencies will have type `jar` by default if no value is specified? Well here we know that `helidon-dependencies` is a BOM thus it does not have a JAR file associated with it. If we leave out the type element then Apache maven will complain with a warning and will fail to resolve the version of `netty-handler`, so be sure to not to miss setting this element correctly. Resolving the dependency graph yields the following result:

```

$ mvn dependency:tree
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.acme:example >-----
-----
[INFO] Building example 1.0.0-SNAPSHOT
[INFO] -----[ jar ]-----

```

```

-----
[INFO]
[INFO] --- maven-dependency-plugin:2.8:tree (default-cli) @ example ---
[INFO] com.acme:example:jar:1.0.0-SNAPSHOT
[INFO] \- io.netty:netty-handler:jar:4.1.51.Final:compile
[INFO]    +- io.netty:netty-common:jar:4.1.51.Final:compile
[INFO]    +- io.netty:netty-resolver:jar:4.1.51.Final:compile
[INFO]    +- io.netty:netty-buffer:jar:4.1.51.Final:compile
[INFO]    +- io.netty:netty-transport:jar:4.1.51.Final:compile
[INFO]    \- io.netty:netty-codec:jar:4.1.51.Final:compile
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
-----

```

We can see that the correct version was chosen and that every direct dependency of `netty-handler` is resolved as transitive as well.

The third and final purpose of the `<dependencyManagement>` section comes into play when there is a parent-child relationship between POMs. The POM format defines a `<parent>` section that any POM can use to establish a link with another POM that is seen as a parent. Parent POMs provide configuration that can be inherited by child POMs, the parent `<dependencyManagement>` section being one of them. Apache Maven follows the parent link upwards until it can no longer find a parent definition, then processes down the chain resolving configuration, with the POMs located at lower levels overriding configuration set by those POMs in higher levels. This means a child POM always has the option to override configuration declared by a parent POM. Thus, a `<dependencyManagement>` section found in the parent POM will be visible to the child POM, as if it were defined on the child. We still get the same benefits from the two previous purposes of this section, which means we can fix versions for transitive dependencies and import BOM dependencies. The following is an example of a parent POM that declares `helidon-dependencies` and `commons-lang3` in its own `<dependencyManagement>` section:

```

<?xml version="1.0" encoding="UTF-8"?>
<project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd"
    xmlns="http://maven.apache.org/POM/4.0.0"

```

```

    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<modelVersion>4.0.0</modelVersion>
<groupId>com.acme</groupId>
<artifactId>parent</artifactId>
<version>1.0.0-SNAPSHOT</version>
<packaging>pom</packaging>

<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>io.helidon</groupId>
      <artifactId>helidon-dependencies</artifactId>
      <version>2.2.0</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
    <dependency>
      <groupId>org.apache.commons</groupId>
      <artifactId>commons-lang3</artifactId>
      <version>3.11</version>
    </dependency>
  </dependencies>
</dependencyManagement>
</project>

```

Given the fact that there is no JAR file associated with this POM file we also must explicitly define the value for the `<packaging>` element as `pom`. The child POM requires the use of the `<parent>` element to refer to this POM, shown in the following example:

```

<?xml version="1.0" encoding="UTF-8"?>
<project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd"
  xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>com.acme</groupId>
    <artifactId>parent</artifactId>
    <version>1.0.0-SNAPSHOT</version>
  </parent>
  <artifactId>example</artifactId>

  <dependencies>
    <dependency>
      <groupId>io.netty</groupId>
      <artifactId>netty-handler</artifactId>

```



```

    </dependency>
    <dependency>
      <groupId>org.apache.commons</groupId>
      <artifactId>commons-lang3</artifactId>
    </dependency>
  </dependencies>
</project>

```

Perfect! With this setup ready it's time to once again resolve the dependency graph and inspect its contents.

```

$ mvn dependency:tree
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.acme:example >-----
-----
[INFO] Building example 1.0.0-SNAPSHOT
[INFO] -----[ jar ]-----
-----
[INFO]
[INFO] --- maven-dependency-plugin:2.8:tree (default-cli) @ example ---
-
[INFO] com.acme:example:jar:1.0.0-SNAPSHOT
[INFO] +- io.netty:netty-handler:jar:4.1.51.Final:compile
[INFO] |   +- io.netty:netty-common:jar:4.1.51.Final:compile
[INFO] |   +- io.netty:netty-resolver:jar:4.1.51.Final:compile
[INFO] |   +- io.netty:netty-buffer:jar:4.1.51.Final:compile
[INFO] |   +- io.netty:netty-transport:jar:4.1.51.Final:compile
[INFO] |   \- io.netty:netty-codec:jar:4.1.51.Final:compile
[INFO] \- org.apache.commons:commons-lang3:jar:3.11:compile
[INFO] -----
-----
[INFO] BUILD SUCCESS
[INFO] -----
-----

```

We've got two direct dependencies as expected, with the correct GAV coordinates, as well as the transitive dependencies as seen earlier. There are a few additional items related to dependency management and resolution such as dependency exclusions (eject a transitive dependency by its GA coordinates), failing the build on dependency conflicts (different versions of the same GA coordinates found in the graph), and more. However it's best to stop here and have a look at what Gradle offers in terms of dependency management.

Dependency management with Gradle

Dependency management with Gradle

As mentioned earlier Gradle builds on top of the lessons learned from Apache Maven and understands the POM format, allowing it to provide dependency resolution capabilities similar to Apache Maven. However it also adds additional capabilities and finer grained control. This section will refer to topics already covered in the previous section, so I recommend you to read it first in the event you skipped it, or if you need a refresher on dependency management as provided by Apache Maven. Let's have a look at what Gradle offers.

First off, you must select the DSL for writing the build file. Your options are the Apache Groovy DSL or the Kotlin DSL. We'll continue with the former as the Apache Groovy DSL is the one that has more examples found in the wild. It's also easier to move from Apache Groovy to Kotlin than vice-versa, meaning that snippets written with the Groovy DSL can be used verbatim with the Kotlin DSL (with perhaps a few changes suggested by the IDE) whereas moving in the other direction requires knowledge of both DSLs. The next step is picking the format for recording dependencies, for which there are quite a few; the most common formats are a single literal with GAV coordinates such as:

```
'org.apache.commons:commons-collections4:4.4'
```

And the Map literal that splits each member of the GAV coordinates into its own element, such as

```
group: 'org.apache.commons', name: 'commons-collections4', version: '4.4'
```

Note that Gradle chose to go with `group` instead of `groupId` and `name` instead of `artifactId` however the semantics are the same. The next order of business is declaring dependencies for a particular scope (in Apache Maven's terms) however Gradle calls this "configuration" and the behavior goes beyond what scopes are capable of. Assuming the `java-library` plugin is applied to a Gradle build file we gain access to the following configurations by default:

- **api**: Defines dependencies required for compiling production code and affects the compile classpath. It is equivalent to the `compile` scope and thus is mapped as such when a POM is generated.

- **implementation:** Defines dependencies required for compilation but are deemed as implementation details; they are more flexible than dependencies found in the api configuration. This configuration affects the compile classpath but will be mapped to the runtime scope when a POM is generated.
- **compileOnly:** Defines dependencies required for compilation but not for execution. This configuration affects the compile classpath but these dependencies are not shared with other classpaths. Also, they are not mapped to the generated POM.
- **runtimeOnly:** Dependencies in this configuration are needed for execution only and only affect the runtime classpath. They are mapped to the runtime scope when a POM is generated.
- **testImplementation:** Defines dependencies required for compiling test code and affects the testCompile classpath. They are mapped to the test scope when a POM is generated.
- **testCompileOnly:** Defines dependencies required for compiling test code but not for execution. This configuration affects the test compile classpath but these dependencies are not shared with the test runtime classpath. Also, they are not mapped to the generated POM.
- **testRuntimeOnly:** Dependencies with this configuration are needed for executing test code and only affect the testRuntime classpath. They are mapped to the test scope when a POM is generated.

You may see additional configurations depending on the Gradle version in use, including the following legacy ones (which were deprecated in Gradle 6 and removed in Gradle 7):

- **compile:** This configuration was split into api and implementation.
- **runtime:** Deprecated in favor of runtimeOnly.
- **testCompile:** Deprecated in favor of testImplementation to align with the implementation configuration name.
- **testRuntime:** Deprecated in favor of testRuntimeOnly to be consistent

with `runtimeOnly`.

Similarly as with Apache Maven the classpaths follow a hierarchy. The compile classpath can be consumed by the runtime classpath, thus every dependency set in the api or implementation configurations are also available for execution. This classpath can also be consumed by the test compile classpath, enabling production code to be seen by test code. The runtime and test classpaths are consumed by the test runtime classpath, allowing test execution access to all dependencies defined in all the configurations so far mentioned.

Like with Apache Maven, dependencies can be resolved from repositories. Unlike Apache Maven where both Maven local and Maven Central repositories are always available, in Gradle we must explicitly define the repositories from which dependencies may be consumed. Gradle lets you define repositories that follow the standard Apache Maven layout, the Ivy layout, and even local directories with a flat layout. It also provides conventional options to configure the most commonly known repository, Maven Central. We'll use `mavenCentral` for now as our only repository. Putting together everything we have seen so far, we can produce a build file like the following:

```
plugins {  
    id 'java-library'  
}  
  
repositories {  
    mavenCentral()  
}  
  
dependencies {  
    api 'org.apache.commons:commons-collections4:4.4'  
}
```

We can print out the resolved dependency graph by invoking the `dependencies` task, however this will print out the graph for every single configuration, thus we'll print out only the resolved dependency graph for the compile classpath for the sake of keeping the output short and also to showcase an extra setting that can be defined for this task.

```
$ gradle dependencies --configuration compileClasspath  
  
> Task :dependencies
```

```
-----  
Root project  
-----
```

```
compileClasspath - Compile classpath for source set 'main'.  
\--- org.apache.commons:commons-collections4:4.4
```

As we can see, only a single dependency is printed out, as `commons-collections` does not have any direct dependencies of its own that are visible to consumers. Let's see what happens when we configure another dependency that brings in additional transitive dependencies, but this time using the `implementation` configuration which will show that both `api` and `implementation` contribute to the compile classpath. The updated build file looks like this:

```
plugins {  
    id 'java-library'  
}  
  
repositories {  
    mavenCentral()  
}  
  
dependencies {  
    api 'org.apache.commons:commons-collections4:4.4'  
    implementation 'commons-beanutils:commons-beanutils:1.9.4'  
}
```

Running the `dependencies` task with the same configuration as before now yields the following result:

```
$ gradle dependencies --configuration compileClasspath
```

```
> Task :dependencies
```

```
-----  
Root project  
-----
```

```
compileClasspath - Compile classpath for source set 'main'.  
+--- org.apache.commons:commons-collections4:4.4  
\--- commons-beanutils:commons-beanutils:1.9.4  
    +--- commons-logging:commons-logging:1.2  
        \--- commons-collections:commons-collections:3.2.2
```

```
\--- commons-collections:commons-collections:3.2.2
```

This tells us our consumer project has two direct dependencies contributing to the compile classpath, and that one of those dependencies brings two additional dependencies seen as transitive from our consumer's point of view. If for some reason you'd like to skip bringing those transitive dependencies into your dependency graph then you can add an extra block of configuration on the direct dependency that declares them, such as:

```
plugins {  
    id 'java-library'  
}  
  
repositories {  
    mavenCentral()  
}  
  
dependencies {  
    api 'org.apache.commons:commons-collections4:4.4'  
    implementation('commons-beanutils:commons-beanutils:1.9.4') {  
        transitive = false  
    }  
}
```

Running the `dependencies` task once more now shows only the direct dependencies and no transitive dependencies.

```
$ gradle dependencies --configuration compileClasspath
```

```
> Task :dependencies
```

```
-----  
Root project  
-----
```

```
compileClasspath - Compile classpath for source set 'main'.  
+--- org.apache.commons:commons-collections4:4.4  
\--- commons-beanutils:commons-beanutils:1.9.4
```

The final aspect I'd like to cover before moving on to the next section is the fact that unlike Apache Maven, Gradle understands semantic versioning and will act accordingly during dependency resolution, choosing the highest version number as a result. We can verify this by configuring two different versions of the same dependency, no matter if they are direct or transitive, as shown in the following

dependency, no matter if they are direct or transitive, as shown in the following snippet:

```
plugins {  
    id 'java-library'  
}  
  
repositories {  
    mavenCentral()  
}  
  
dependencies {  
    api 'org.apache.commons:commons-collections4:4.4'  
    implementation 'commons-collections:commons-collections:3.2.1'  
    implementation 'commons-beanutils:commons-beanutils:1.9.4'  
}
```

In this case we have declared a direct dependency for commons-collections version 3.2.1. We know from previous runs that commons-beanutils:1.9.4 brings in version 3.2.2 of commons-collections, given that 3.2.2 is considered more recent than 3.2.1, we expect that 3.2.2 will be resolved. Invoking the dependencies task yields the following result:

```
$ gradle dependencies --configuration compileClasspath  
  
> Task :dependencies  
  
-----  
Root project  
-----  
  
compileClasspath - Compile classpath for source set 'main'.  
+--- org.apache.commons:commons-collections4:4.4  
+--- commons-collections:commons-collections:3.2.1 -> 3.2.2  
\--- commons-beanutils:commons-beanutils:1.9.4  
    +--- commons-logging:commons-logging:1.2  
    \--- commons-collections:commons-collections:3.2.2  
  
(*) - dependencies omitted (listed previously)
```

As expected, version 3.2.2 was selected. The output even contains an indicator telling us when a dependency version was set to a different value other than the requested one. Versions may be also configured to be fixed regardless of their semantic versioning scheme, even to lower values. This is due to the fact that

Gradle offers more flexible options for dependency resolution strategies, however that falls into the realm of advanced topics alongside dependency locking, strict vs. suggested versions, dependency relocation, platforms and enforced platforms (Gradle's way to interact with BOM artifacts), and more.

Dependency management basics for containers

Further along the line of the software development cycle, you may encounter a step where packaging your maven and gradle project into a container image is necessary. Just like other dependencies in your project, your container images must also be managed appropriately and in concert with other required artifacts. Containers are discussed in detail in Chapter 2, but this section will focus primarily on some of the subtleties of container image management. Just like the dependency management in the automated build tools Maven and Gradle, there be even more dragons ahead!

As you learned in Chapter 2, containers are launched using container images which are most commonly defined using a Dockerfile. The Dockerfile does the work of defining each layer of the image that will be used to build the running container. From this definition, you will get a base distribution layer, code libraries and frameworks, and any other needed files or artifacts that are required to run your software. Here you will also define any necessary configuration (e.g. open ports, database credentials, references to messaging servers, etc) as well as any required users and permissions.

It's line one of the Dockerfile that we will discuss in this section first, or in the case of multi-stage build Dockerfiles, the lines that begin with the directive `FROM`. Similarly to the Maven POM, an image can be built from a `parent` image which may be built from another parent image — a hierarchy that goes all the way to a `base` image, the original ancestor. It is here that we must pay special attention to how our images are composed.

If you remember from Chapter 2, the versioning of docker images is intended to provide flexibility during the development stage of your software as well as a measure of confidence that you are using the latest maintenance update of an image when desired. Most often this is done by referencing the special image version `latest` (the default version if a version is not specified), a request for

which will retrieve what is assumed to be the latest version of the image that is in active development. Although not a perfect comparison, this is most like using a `snapshot` version of a Java dependency. This is all fine and good during development, but when it comes to troubleshooting a new bug in production, this type of versioning in a production image artifact can make troubleshooting more of a challenge. Once an image has been built with this default `latest` version in place for a parent or a base image, it may be very difficult or even impossible to reproduce the build. Just like you would want to avoid using a snapshot dependency in a production release, I would recommend locking down your image versions and avoid using the default `latest` in order to limit the number of moving parts.

Simply locking down your image versions isn't sufficient from the context of security. Use only trusted base images when building your containers. This tip might seem like an obvious one, but third-party registries often don't have any governance policies for the images stored in them. It's important to know which images are available for use on the Docker host, understand their provenance, and review the content in them. You should also enable Docker Content Trust (DCT) for image verification and install only verified packages into images.

Use minimal base images that don't include unnecessary software packages that could lead to a larger attack surface. Having fewer components in your container reduces the number of available attack vectors, and a minimal image also yields better performance because there are fewer bytes on disk and less network traffic for images being copied. BusyBox and Alpine are two options for building minimal base images. Pay just as careful attention to any additional layers you build on top of your verified base image by explicitly specifying all versions of software packages or any other artifacts you pull into the image.

Artifact Publication

Up to now we've discussed how artifacts and dependencies may be resolved, often times from locations known as repositories but we have yet to elaborate what is a repository and how you may publish artifacts to it. In the most basic sense an artifact repository is a file storage that keeps track of artifacts. A repository collects metadata on each published artifact and uses said metadata to offer additional capabilities such as search, archiving, ACLs, and others. Tools

can harness these metadata to offer other capabilities on top such as vulnerability scanning, metrics, categorization, and more.

There are two types of repositories in the case of Maven dependencies, those dependencies that can be resolved via GAV coordinates: local and remote. Apache Maven uses a configurable directory found in the local file system to keep track of dependencies that have been resolved. These dependencies may have been downloaded from a remote repository or may have been directly placed there by the Apache Maven tool itself. This directory is typically referred to as “Maven Local” and its default location is `.m2/repository` found at the home directory of the current user. This location is configurable. On the other side of the spectrum are remote repositories; they are handled by repository software such as Sonatype Nexus, JFrog Artifactory, and others. The most well known remote repository is Maven Central, which is the canonical repository for resolving artifacts.

Let’s discuss now how we can publish artifacts to local and remote repositories.

Publishing to Maven Local

Apache Maven offers 3 ways for publishing artifacts to the Maven Local repository. Two of them are explicit and one is implicit. We’ve already covered the implicit one, it occurs whenever Apache Maven resolves a dependency from a remote repository, as a result a copy of the artifact and its metadata (the associated `pom.xml`) will be placed in the Maven Local repository. This behavior occurs by default as Apache Maven uses Maven Local as a cache to avoid requesting artifacts over the network all over again.

The other two ways of publishing artifacts to Maven Local are by explicitly “installing” files into the repository. Apache Maven has a set of life cycle phases of which *install* is one of them. These phases are well known by Java developers as they are used (and abused) to compile, test, package, and install artifacts to Maven Local. The Apache Maven lifecycle phases follow a predetermined sequence:

Available lifecycle phases are: `validate`, `initialize`, `generate-sources`, `process-sources`, `generate-resources`, `process-resources`, `compile`, `process-classes`, `generate-test-sources`, `process-test-sources`, `generate-test-resources`, `process-test-resources`, `test-compile`, `process-test-classes`, `test`, `prepare-package`, `package`, `pre-integration-test`, `integration-test`, `post-integration-test`, `verify`, `install`.

test, integration-test, post-integration-test, verify, install, deploy, pre-clean, clean, post-clean, pre-site, site, post-site, site-deploy.

Phases are executed in sequence order until the terminal phase is found. Thus invoking *install* typically results in an almost full build (except for *deploy* and *site*). I mentioned *install* is abused as most times is enough to invoke *verify*, the phase that's right before *install*, as the former will force compile, test, package, and integration-test but does not pollute Maven Local with artifacts if they are not needed. This is in no way a recommendation to drop *install* in favor of *verify* all the time as there are use cases when a test will require resolving artifacts from Maven Local. Bottom line we have to be aware of the inputs/outputs of each phase and their consequences.

Back to installing, the first way to install artifacts to Maven Local is simply invoking the *install* phase, like so

```
$ mvn install
```

This will place copies of all `pom.xml` files renamed to follow `<artifactId>-<version>.pom` convention as well as every attached artifact into Maven Local. Attached artifacts are typically the binary JARs produced by the build but can also include other JARs such as the `-sources` and `-javadoc` JARs. The second way to install artifacts is by manually invoking the `install:install-file` goal with a set of parameters. Let's say you have a JAR (`artifact.jar`) and a matching POM file (`artifact.pom`), installing them can be done in the following way:

```
$ mvn install:install-file -Dfile=artifact.jar -DpomFile=artifact.pom
```

Apache Maven will read the metadata found in the POM file and place the files in their corresponding location based on the resolved GAV coordinates. It's possible to override the GAV coordinates, generate the POM on the fly, even omit the explicit POM if the JAR contains a copy inside (that's typically the case for JARs built with Apache Maven, Gradle on the other hand does not include the POM by default).

Now in the case of Gradle there is one way to publish artifacts to Maven Local

and that is by applying the `maven-publish` plugin. This plugin adds new capabilities to the project such as the `publishToMavenLocal` task, as the name implies this will copy the built artifacts and a generated pom to Maven Local. Opposed to Apache Maven, Gradle does not use Maven Local as a cache as it has its own caching infrastructure, thus when Gradle resolved dependencies the files will be placed at a different location, usually `.gradle/caches/modules-2/files-2.1` located at the home directory of the current user.

This covers publishing to Maven Local, now let's have a look at remote repositories.

Publishing to Maven Central

The Maven Central repository is the backbone that allows Java projects to be built on a daily basis. The software running Maven Central is Sonatype Nexus, an artifact repository provided by Sonatype. Given its important role in the Java ecosystem Maven Central has placed a set of rules that must be followed when publishing artifact; Sonatype has published a [guide](#) explaining the pre-requisite and rules. I highly recommend reading through the guide in case requirements have been updated since the publication of this book. In short, you must ensure the following

- You can prove ownership of the reverse domain of the target `groupId`. If your `groupId` is `com.acme.*` then you must own `acme.com`.
- When publishing a binary JAR you must also supply a `-sources` and `-javadoc` JARs, as well as a matching POM, that is, a minimum of 4 separate files.
- When publishing an artifact of type POM then only the pom file is needed.
- PGP signature files for all artifacts must be submitted as well. The PGP keys used for signing must be published in public key servers to let Maven Central verify signatures.
- And perhaps the aspect that trips most people at the beginning, POM

files must comply with a minimum set of elements such as `<license>`, `<developers>`, `<scm>`, and others. These elements are described in the guide; the omission of any of them will cause a failure during publication and as result artifacts won't be published at all.

We can avoid the last problem or at least detect much earlier during development by using the **pomchecker** project. Pomchecker can be invoked in many ways: as a standalone CLI tool, as a Apache Maven plugin, or as Gradle plugin. The different ways to invoke pomchecker make it ideal for verifying a POM at your local environment or at a CI/CD pipeline. Verifying a `pom.xml` file using the CLI can be done like so

```
$ pomchecker check-maven-central --pom-file=pom.xml
```

If your project is built with Apache Maven then you may invoke the pomchecker plugin without having to configure it in the pom, like this

```
$ mvn org.kordamp.maven:pomchecker-maven-plugin:check-maven-central
```

This command will resolve the latest version of the `pomchecker-maven-plugin` and execute its `check-maven-central` goal right on the spot, using the current project as input. In the case of Gradle you'd have to explicitly configure the `org.kordamp.gradle.pomchecker` plugin as Gradle does not offer an option for invoking an inlined plugin like Apache Maven does.

The last bit of configuration that must be applied to the build is the publication mechanism itself, this can be done by adding the following to your `pom.xml` if you're building with Apache Maven:

```
<distributionManagement>
  <repository>
    <id>ossrh</id>

    <url>https://s01.oss.sonatype.org/service/local/staging/deploy/maven2/
    </url>
  </repository>
  <snapshotRepository>
    <id>ossrh</id>

    <url>https://s01.oss.sonatype.org/content/repositories/snapshots</url>
```

```

</snapshotRepository>
</distributionManagement>

<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-javadoc-plugin</artifactId>
      <version>3.2.0</version>
      <executions>
        <execution>
          <id>attach-javadocs</id>
          <goals>
            <goal>jar</goal>
          </goals>
          <configuration>
            <attach>true</attach>
          </configuration>
        </execution>
      </executions>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-source-plugin</artifactId>
      <version>3.2.1</version>
      <executions>
        <execution>
          <id>attach-sources</id>
          <goals>
            <goal>jar</goal>
          </goals>
          <configuration>
            <attach>true</attach>
          </configuration>
        </execution>
      </executions>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-gpg-plugin</artifactId>
      <version>1.6</version>
      <executions>
        <execution>
          <goals>
            <goal>sign</goal>
          </goals>
          <phase>verify</phase>
          <configuration>
            <gpgArguments>
              <arg>--pinentry-mode</arg>
            </gpgArguments>
          </configuration>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>

```

```

        <arg>loopback</arg>
      </gpgArguments>
    </configuration>
  </execution>
</executions>
</plugin>
<plugin>
  <groupId>org.sonatype.plugins</groupId>
  <artifactId>nexus-staging-maven-plugin</artifactId>
  <version>1.6.8</version>
  <extensions>true</extensions>
  <configuration>
    <serverId>central</serverId>
    <nexusUrl>https://s01.oss.sonatype.org</nexusUrl>
    <autoReleaseAfterClose>true</autoReleaseAfterClose>
  </configuration>
</plugin>
</plugins>
</build>

```

Note that this configuration generates `-sources` and `-javadoc` JARs, signs all attached artifacts with PGP, and uploads all artifacts to the given URL which happens to be one of the URLs supported by Maven Central. The `<serverId>` element identifies the credentials you must have in place in your `settings.xml` file, otherwise the upload will fail; or you may define credentials as command line arguments. You might want to put the plugin configuration inside a `<profile>` section as the behavior provided by the configured plugins is only needed when a release is posted, no need to generate additional JARs during the execution of the main lifecycle phase sequence, this way your builds will only execute the minimum set of steps and thus be faster as a result. On the other hand if you're using Gradle for publication you'd have to configure a plugin that can publish to a Sonatype Nexus repository, the latest of such plugins being [io.github.gradle-nexus.publish-plugin](https://github.com/gradle-nexus/publish-plugin). There's more than one way to configure Gradle to do the job, idioms change more rapidly than what you must configure in Apache Maven, I'd recommend you to consult the official Gradle guides to find out what needs to be done in this case.

Publishing to Sonatype Nexus

You may recall that Maven Central is run using Sonatype Nexus thus it should be no surprise that the configuration shown in the previous section applies here as well. In fact you only have to change the publication URLs to match

as well, matter of fact you only have to change the publication URLs to match the Nexus repository. There's one caveat though, the strict verification rules applied by Maven Central often times do not apply to a custom Nexus installation. That is, Nexus has the option to configure the rules that govern artifact publication. These rules may be relaxed for a Nexus instance running within your organization for example, or they may be stricter in other areas. It's a good idea to consult the documentation available at your organization regarding artifact publication to their own Nexus instance.

One thing is clear though, if you are publishing artifacts to your organization's Nexus repository that eventually must be published to Maven Central then it's a good idea to follow the Maven Central rules from the start, of course as long as these rules do not clash with your organization's.

Publishing to JFrog Artifactory

JFrog Artifactory is another popular option for artifact management. It offers similar capabilities as Sonatype Nexus while at the same time adding other features such as integration with other products that belong to the JFrog Platform such as X-Ray and Pipelines. One particular feature I'm quite fond of is that artifacts do not need to be signed at the source before publication, Artifactory can perform the signing with your PGP keys or with a site wide PGP key. This relieves you of the burden of setting up keys on your local and CI environments as well as transferring less bytes during publication. As before, the previous publication configuration we saw for Maven Central will work for Artifactory as well, by only change the publication URLs to match the Artifactory instance.

As it is the case with Nexus, Artifactory also allows you to sync artifacts to Maven Central, in which you have to follow the rules for publishing to Maven Central once again, thus publishing well formed POMs, sources & javadoc JARs from the get go is a good idea.

¹ The Surprising Power of Online Experiments Getting the most out of A/B and other controlled tests by Ron Kohavi and Stefan Thomke. <https://hbr.org/2017/09/the-surprising-power-of-online-experiments>

Chapter 6. Securing Your Binaries

Sven Ruppert

Stephen Chin

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors’ raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 7th chapter of the final book. If there is a GitHub repo associated with the book, it will be made active after final publication.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at ccollins@oreilly.com.

Data is the pollution problem of the information age, and protecting privacy is the environmental challenge.

—Bruce Schneier, *Data and Goliath: The Hidden Battles to Collect Your Data and Control Your World*

Software security is a critical part of any comprehensive DevOps rollout. It has become highlighted by multiple recent breaches and become the subject of attention by new government regulations. The impact is across the entire software lifecycle from development through to production. As a result, DevSecOps is something that every software developer and DevOps professional needs to understand.

In this chapter we will introduce you to the fundamentals of security analysis and vulnerability mitigation. You will learn how to evaluate your product and organizational risk for security vulnerabilities. We will also cover static and dynamic techniques for security testing and scoring techniques for risk assessment. Regardless of your role you will be better prepared to help secure your organization’s software delivery lifecycle.

But first let's look deeper into what happens if you don't have a focus on security and secure your software supply chain.

Supply Chain Security Compromised

It started in early December 2020 when the FireEye company noticed that it had become a victim of a cyber attack. The remarkable thing about this is that FireEye itself specializes in detecting and fending off cyber attacks. The internal analysis showed that the attackers managed to steal the FireEye internal tools, which they use to examine their customer's IT infrastructure for weak points. It was a highly specialized toolbox optimized for breaking into networks and IT systems of all kinds. This toolbox in the hands of hackers is a big problem in itself, but at the time of the breach, the intended use was not known. It took some time to identify the link to further cyber attack, one of which became known as the "SolarWinds Hack."

What Happened at SolarWinds?

SolarWinds is a company based in the United States that specializes in the management of complex IT network structures with its software products. For this, they have developed a product known as the "Orion Platform". The company itself has over 300,000 active customers who use this software internally. The software for managing network components has to be equipped with very generous administrative rights within the IT systems in order to be able to carry out its tasks. And here is one of the critical points the hackers used in their strategy.

It took some time to recognize the connection between the FireEye hack and the later, massive cyber attacks, because the chain of effects was not as direct as previous vulnerability breaches.

Due to the long gap between exploitation of this vulnerability and discovery of the breach, many companies and government organizations ended up being affected by this attack. Over a period of a few weeks 20,000 successful attacks were launched. Because the pattern of the attacks was similar, security researchers were able to identify that these attacks were related. One of the common characteristics was that all of the organizations who suffered an attack used

SolarWinds software to manage their network infrastructure.

The attackers who appropriated the tools of the FireEye company used these tools to break into the networks of the SolarWinds company. Here they had the aim of attacking the continuous integration pipeline, which is responsible for creating the binaries for the Orion software platform. The production line was manipulated in such a way that each time a new version was run through, the resulting binary was compromised. They were able to produce binaries that included the a backdoor prepared by the hackers. The product was used here as a carrier ship, and the delivery took place via the distribution channels of the SolarWinds company, which gave the compromised binaries unhindered access to thousands of networks. The challenge lies in the fact that the recipient checks when using the fingerprints supplied by the producer, which signals a faultless transmission. There is also a specific relationship of trust between the customer and the SolarWinds company. And precisely at this point was one of the significant weaknesses of the victims of this cyber attack.

The course of the attack was then as follows. The company, SolarWinds, creates an update of its software and makes these binaries available to all 300,000 customers via an automatic update process. Almost 20,000 customers then installed this update in a short period of time. The compromised software waited about two weeks after activation and then began to spread in the infected systems. As if that wasn't bad enough, over time, further malware was then dynamically loaded, making it almost impossible to follow up afterwards. Now, even if you now closed the first vulnerability gateway, it became impossible to know how many other gateways had been opened in the affected systems. At this point, the only way to repair the compromised system is to rebuild it from scratch.

At this point, lets differentiate between the perspective of the SolarWinds company, versus the perspective of the affected customers. Whose responsibility is it to mitigate this attack, and what does the procedure look like if you are affected yourself? Which tools can you use to identify and address the vulnerability? Who can take action against such attacks, and at what point in the vulnerability timeline?

Security from the Vendor Perspective

Figure 1.1: Security from the Vendor Perspective

First, let's start with the perspective of a software manufacturer who distributes software to their customers. With the new way the attacks are carried out, you have to prepare yourself because you will only be the carrier. This unique quality of attacks now requires a very stringent approach in your own software development and distribution process.

Securing the tools used is one of the most important aspects, because they have access to your internal systems and can maliciously modify binaries in your software pipeline. Securing tools is challenging, because it is challenging to examine all the tools in your software delivery lifecycle.

Security from the Customer Perspective

As a customer it is essential to consider all elements in the value chain, including all of the tools that a software developer uses on a daily use. You also have to check the binaries generated from your CI/CD system for the possibility of modification or vulnerability injection. Here it is essential to keep an overview of all components used with a secure and traceable bill of materials. Ultimately, it only helps if you break down your own products into their constituent parts and subject each element to a security review.

How can you protect yourself as a consumer? The approach that all elements must be subjected to a critical review also applies here. As has been seen, individual fingerprints and the exclusive use of confidential sources do not provide the best possible protection. The components used must be subjected to a "deep inspection".

The Full Impact Graph

The SolarWinds hack demonstrates the need to analyze the full impact graph in order to discover vulnerabilities in your supply chain. If you now find a vulnerability in a binary file, then the relevance of this vulnerability is in the context of use. You need to know where this file is used, and the final risk that is caused by this weak point is a result of the operational environment. If you don't use this binary anywhere, then it can't do any harm. However, if the use occurs in the critical areas within a company, then there is a calculable risk.

Full Impact Graph, represents all involved areas of application of the known vulnerability. An example would be a jar file that is packaged in a war file in a

Docker layer and then provisioned in the production environment using the HELM Chart. This requires tools that are used to check for known weak points. These tools can only develop their full potential if they can recognize and represent the interrelationships across technology boundaries. The focus on just one technology can very quickly lead to pseudo-security that can become dangerous.

At this point, I would like to anticipate a little and point out that the CVSS provides an environmental metric precisely for assessing one's own context. But more on that in the section on CVSS.

Securing your DevOps infrastructure

Now that we understand the impact of security vulnerabilities, it is time to look at countermeasures we can utilize. First, let's shed some light on the procedures and roles that are used in a DevOps environment.

The Rise of DevSecOps

I won't go into all the details that led to a merger between Dev and Ops to become DevOps. However, I would like to briefly return to the key points here, as these play a central role in introducing security.

It has been recognized that the two areas of Dev and Ops have to work closer together in order to become more productive. If you read the description on Wikipedia, you can see that the steps listed there relate directly to the production steps of software development. There was the transition to production, i.e. to the departments that deal with the operation of the software. The transfer point mainly was a repository in which the software used in each case was kept. Using a repository is still undisputedly the right way to exchange binaries. However, the roles of those involved have changed a little. The skills of the developers were expanded to include the core competencies of the administrators and vice versa. The associated understanding of the needs of the other side helps to optimize all processes involved. But what about security now? Security is not and should never be an explicit step in software development. Safety is a cross-cutting issue that goes through all phases of production through to operation. This, in turn, brings the realization that there is no dedicated safety officer who

can do this work alone. The team as a whole is entrusted with the issue of safety, just as they are, for example, with the issue of quality.

The result of this knowledge can be found under the name DevSecOps. However, there are some subtleties here that cannot be ignored. Not everyone in the production chain can do all things equally well. Everyone has their own idiosyncrasies and is more efficient in some areas. Accordingly, even in a DevSecOps organization, some team members care more about the Dev area and others who have their strengths in the Ops area.

The Role of SREs in Security

An exception to the Dev and Ops specialization is the SRE (Site Reliability Engineer) role. The term originally comes from Google and describes the people in a team who deal with the reliability of services. The metric against which an SRE works is called the error budget. It is assumed that the software has errors and that this is exactly what leads to downtimes. For this, there is a specific error budget, or downtime budget, for a service. The SRE aims to get by with the defined budget and to really see these times as a kind of budget that can be invested. Investments in the form of downtime can therefore serve to improve the system. However, downtime due to bugs, damage or cyber-attacks is also deducted from this budget.

Therefore, an SRE is a team member that should ensure the balance between the robustness of the systems and the introduction of new features. For this purpose, he is given up to a maximum of 50 per cent of his working time to deal practically with the field of Ops. This time should be used to automate the systems more and more. The rest of the working time is an SRE working as a developer and involved in implementing new features. And now we come to the exciting question; Is an SRE also responsible for security?

DevSecOps and SRE are differentiated by the fact that DevSecOps is a process model and SRE is a role. This role of an SRE is in the middle of a DevSecOps structure since the working hours and skills are almost exactly half reflected in the Dev and Ops area. SREs are usually developers with many years of experience who now specialize in the Ops area, or the other way around, an administrator with many years of professional experience and who is now deliberately entering into software development.

And is security a task for the SRE? If you remember the example of SolarWinds again, the question arises of who has the most influence within the value chain to take action against vulnerabilities. For this purpose, we will take a closer look at the two areas Dev and Ops and the options available there.

Static and Dynamic Security Analysis

There are two main types of security analysis abbreviated as SAST and DAST. We will now briefly examine what these mean and how the two approaches differ.

Static Application Security Testing

The abbreviation SAST stands for “Static Application Security Testing” and is a procedure that focuses on the static analysis of the application. The focus here is on recognizing and localizing the known vulnerabilities.

How Static Application Security Testing works

SAST is a so-called white box process. For this procedure, you need to have access to the source code of the application to be tested. However, an operational runtime environment does not have to be available. The application does not have to be executed for this procedure, which is why the term static is also used. Three types of security threats can be identified using SAST.

1. Are there gaps in the source code in the functional area that allows, for example, “tainted code” to be smuggled in? These are lines that can later infiltrate malware.
2. Are there any source code lines that allow you to connect to files or certain object classes? The focus here is also on detecting and preventing the introduction of malware.
3. Are there gaps on the application level that allow you to interact with other programs unnoticed?

However, it should be noted here that the analysis of the source code is itself a complex matter. The area of static security analysis also includes the tools that enable you to determine and evaluate all contained direct and indirect

enable you to determine and evaluate all contained direct and indirect dependencies. But more on that later when it comes to the “quick wins” in the area of security. As a rule, various SAST tools should check the source code at regular intervals. The source code scanners must also be adapted to your organizational needs with an initial implementation to adjust the scanner to your respective domain. The Open Web Application Security Project (OWASP) Foundation offers assistance; it not only lists typical security vulnerabilities, but also gives recommendations for suitable SAST tools.

We will deal with the most common vulnerabilities again in a later section and explain them briefly.

Advantages of the SAST approach

In comparison with subsequent security tests, the static security analysis approach offers the following advantages:

1. Because the detection already takes place in the development phase, removing the weak points can be carried out much more cost-effectively compared to detection that only takes place at runtime. By accessing the source code, you can also understand how this vulnerability came about and prevent it from recurring in the future. These findings cannot be obtained using a black-box process.
2. Another advantage is partial analysis, which means that even non-executable source text can be analyzed. The static security analysis can be carried out by the developers themselves, which significantly reduces the necessary availability of security experts.

A 100% analysis of the system at the source code level is also possible, which cannot be guaranteed with a dynamic approach. Black box systems can only perform penetration tests, which are an indirect analysis.

Disadvantages of the SAST approach

In theory, the SAST approach does not have any real disadvantages. In practice, however, there are sometimes three fundamental problems.

1. The programming work often suffers, which in turn manifests itself in bugs. The developers focus too much on the security tests and bug fixes.

2. The tools can be problematic. This happens especially if the scanners have not been individually adapted to your own needs.
3. SAST often replaces the subsequent security tests. However, all problems that are directly related to an application in operation remain undetected.

A little later in the “How much scanning is enough scanning” section, I will show how you can increase the advantages of SAST by analyzing the binaries but avoid the two disadvantages mentioned above.

Dynamic Application Security Testing

The abbreviation DAST stands for “Dynamic Application Security Testing” and describes the security analysis of a running application (usually a running web application). A wide variety of attack scenarios are performed in order to identify as many of the weak points as possible in the application. The term dynamic comes from the necessity that a running application must be available to carry out the tests. It is critical that the test system behaves the same as the production environment. Even minor variations can represent serious differences, including different configurations or upstream load balancers and firewalls.

DAST is a black box process in which the application is only viewed from the outside. The technologies used do not play a role in the type of security check, as the application is only accessed generically and externally. This means that all information that could be obtained from the source code is invisible for this type of test. It is, therefore, possible for the person testing to test for the typical problems with a generic set of tools. The benchmark project of the OWASP offers reasonable assistance for the selection of the scanner for your own project. This evaluates the performance of the individual tools in relation to the specific application background.

Advantages of DAST

The DAST process has the following advantages in addition to the fact that the technology used is irrelevant and the security analysis consequently works in a technology-neutral manner:

- The scanners find errors in the runtime environment in which the test is carried out.
- The rate of false positives is low.
- The tools find faulty configurations in basically functional applications. For example, you can identify performance problems that other scanners cannot.
- The DAST programs can be used in all phases of development and also in a later operation.

DAST scanners are based on the same concepts that real attackers use for their malware. They, therefore, provide reliable feedback on weaknesses. Tests have consistently shown that the majority of DAST tools can identify the top 10 most common threats listed by the OWASP Foundation. In the case of individual attack scenarios, however, things look different again.

Disadvantages of DAST

There are a number of disadvantages to using DAST tools.

- Not readily scalable: The scanners are programmed to carry out specific attacks on functional web apps and can usually only be adapted by security experts with the necessary product knowledge. They, therefore, offer little space for individual scaling.
- Slow scan speed: DAST tools can take several days to complete their analysis. Late in the life cycle: DAST tools find some security gaps very late in the development cycle that could have been discovered earlier via SAST. The costs of fixing the corresponding problems are therefore higher than they should be.
- Based on known bugs: It takes a relatively long time for tools to scan for new types of attacks.

Comparing SAST and DAST

Table 6-1 shows a summary of the differences between the SAST and DAST testing approaches.

Table 6-1. SAST vs. DAST

| SAST | DAST |
|---|---|
| <p>White-Box security testing</p> <ul style="list-style-type: none"> • The tester has access to the underlying framework, design and implementation • The application is tested from inside out • This type of testing represents the developer approach | <p>Black-Box security testing</p> <ul style="list-style-type: none"> • The tester has no knowledge of the technologies or framework that the application is built on • The application is tested from the outside in • This type of testing represents the hacker approach |
| <p>Requires source code</p> <ul style="list-style-type: none"> • SAST doesn't require a deployed application • It analyzes the source code or binary without executing the application | <p>Requires a running application</p> <ul style="list-style-type: none"> • DAST doesn't require source code or binaries • It analyzes by executing the application |
| <p>Find vulnerabilities earlier in the SDLC</p> <ul style="list-style-type: none"> • The scan can be executed as soon as code is deemed feature complete | <p>Finds vulnerabilities toward the end of the SDLC</p> <ul style="list-style-type: none"> • Vulnerabilities can be discovered after the development cycle is complete. |
| <p>Less expensive to fix vulnerabilities</p> <ul style="list-style-type: none"> • Since vulnerabilities are found earlier in the SDLC, it is easier and faster to remediate them. • Finding can often be fixed before the code enters the QA cycle. | <p>More expensive to fix vulnerabilities</p> <ul style="list-style-type: none"> • Since vulnerabilities are found toward the end of the SDLC, remediation often gets pushed into the next development cycle • Critical vulnerabilities may be fixed as an emergency release. |
| <p>Can't discover runtime and environmental issues</p> <ul style="list-style-type: none"> • Since the tool scans static code, it can not discover runtime vulnerabilities. | <p>Can discover runtime and environmental issues.</p> <ul style="list-style-type: none"> • Since the tool uses dynamic analysis on a running application, it is able to find runtime vulnerabilities. |
| <p>Typically supports all kind of software</p> <ul style="list-style-type: none"> • Example include web application, web services, and thick clients | <p>Typically scans only web-apps and web-services</p> <ul style="list-style-type: none"> • DAST is not useful for other types of software. |

If you look at the advantages and disadvantages of these two types of security testing, you can see that these two approaches are not mutually exclusive. On the contrary, these approaches complement each other perfectly. SAST can be used to identify known vulnerabilities. DAST can be used to identify vulnerabilities

to identify known vulnerabilities. DAST can be used to identify vulnerabilities that are not yet known, provided they are based on common vulnerabilities. You also gain knowledge about the overall system if you carry out these tests on the production system. However, as soon as you run DAST on test systems, you lose these capabilities again.

The Common Vulnerability Scoring System

The basic idea behind CVSS is to provide a general classification of the severity of a security vulnerability. CVSS stands for Common Vulnerability Scoring System, which is a general vulnerability rating system. The weak points found are evaluated from various points of view. These elements are weighed against each other to obtain a standardized number between 0 and 10.

A rating system, like CVSS, provides us with a standardized number allows us to evaluate different weak points abstractly and derive follow-up actions from them. The focus here is on standardizing the handling of these weak points. As a result, you can define actions based on the value ranges.

In principle, CVSS can be described so that the probability and the maximum possible damage are related using predefined factors. The basic formula for this is: $\text{risk} = \text{probability of occurrence} \times \text{damage}$

The evaluation in the CVSS is based on various criteria and is called “metrics”. For each metric, one or more values selected from a firmly defined selection option. This selection then results in a value between 0.0 and 10.0, where 0 is the lowest and 10 is the highest risk value. The entire range of values is then subdivided into groups and are labeled “None”, “Low”, “Medium”, “High”, and “Critical”.

These metrics are then divided into three orthogonal areas that are weighted differently from one another called “Basic Metrics”, “Temporal Metrics”, and “Environmental Metrics”. Here, different aspects are queried in each area, which must be assigned a single value. The weighting among each other and the subsequent composition of the three group values gives the final result.

However, all component values that lead to this result are always supplied, ensuring that there is transparency at all times about how these values were derived. In the next section, we will explore these three sub-areas in detail.

CVSS Basic Metrics

The basic metrics form the foundation of the CVSS rating system. The aim is to record technical details of the vulnerability that will not change over time, so the assessment is independent of other changing elements. Different parties can carry out the calculation of the base value. It can be done by the discoverer, the manufacturer of the project or product concerned, or by a party (CERT) charged with eliminating this weak point. One can imagine that, based on this initial decision, the value itself will turn out different since the individual groups pursue different goals.

The base value evaluates the prerequisites that are necessary for a successful attack via this security gap. This is the distinction between whether a user account must be available on the target system for an attack that is used in the course of the attack or whether the system can be compromised without the knowledge about a system user. It also plays a significant role in whether a system is vulnerable over the Internet or whether physical access to the affected component is required.

The base value should also reflect how complex the attack is to carry out. In this case, the complexity relates to the necessary technical steps and includes assessing whether the interaction with a regular user is essential. Is it sufficient to encourage any user to interact, or does this user have to belong to a specific system group (e.g. administrator)? The correct classification is not a trivial process; the assessment of a new vulnerability requires exact knowledge of this vulnerability and the systems concerned.

The basic metrics also take into account the damage that this attack could cause to the affected component. This means the possibility to extract the data from the system, manipulate it, or completely prevent the system's use. The three areas of concern are:

- Confidentiality
- Integrity
- Availability

However, you have to be careful here concerning the weighting of these possibilities. In one case, it can be worse when data is stolen than it is changed.

In another case, the unusability of a component can be the worst damage to be assumed.

The scope metric has also been available since CVSS version 3.0. This metric looks at the effects of an affected component on other system components. For example, a compromised element in a virtualized environment enables access to the carrier system. A successful change of this scope represents a greater risk for the overall system and is therefore also evaluated using this factor. This demonstrates that the interpretation of the values also requires adjusting to one's situation, which brings us to the "temporal" and "environment" metrics.

CVSS Temporal Metrics

The time-dependent components of the vulnerability assessment are brought together in the temporal metrics group. Temporal metrics are unique in that the base value can only be reduced and not increased. The initial rating is intended to represent the worst-case scenario. This has both advantages and disadvantages if you bear in mind that it is during the initial assessment of a vulnerability that there are competing interests.

The elements that change over time influence the "Temporal Metrics". On the one hand, this refers to changes concerning the availability of tools that support the exploitation of the vulnerability. These can be exploits or step-by-step instructions. A distinction must be made whether a vulnerability is theoretical or whether a manufacturer has officially confirmed it. All of these events change the base value.

The influence on the initial evaluation comes about through external framework conditions. These take place over an undefined time frame and are not relevant for the actual basic assessment. Even if an exploit is already in circulation during the base values survey, this knowledge will not be included in the primary assessment. However, the base value can only be reduced by the temporal metrics. This approach takes a little getting used to and is often the subject of criticism. The reason why you decided on it is understandable from the theoretical point of view. The base value is intended to denote the maximum amount of damage.

And this is where a conflict arises. The person or group who has found a security gap tries to set the base value as high as possible. A high severity loophole will

sell for a higher price and receive more media attention. The reputation of the person/group who found this gap increases as a result. The affected company or the affected project is interested in exactly the opposite assessment. Therefore, it depends on who finds the security gap, how the review process should take place and by which body the first evaluation is carried out. This value is further adjusted by the Environmental Metrics.

CVSS Environmental Metrics

In the case of environmental metrics, your own system landscape is used to evaluate the risk of the security gap. This means that the evaluation is adjusted based on the real situation. In contrast to Temporal Metrics, Environmental Metrics can correct the base value in both directions. The environment can therefore lead to a higher classification and must also be constantly adapted to your own environment changes.

Let's take an example of a security hole that has an available patch from the manufacturer. The mere presence of this modification leads to a reduction of the total value in the Temporal Metrics. However, as long as the patch has not been activated in your own systems, the overall value must be drastically corrected upwards again via the Environmental Metrics. The reason for this, is because as soon as a patch is available, it can be used to better understand the security gap and its effects. The attacker has more detailed information that can be exploited, which reduces the resistance of the not yet hardened systems.

===== CONTINUE HERE =====

At the end of an evaluation, the final score is obtained, calculated from the three previously mentioned values. The resulting value is then assigned to a value group. But there is one more point that is often overlooked. In many cases, the final score is simply carried over without individual adjustments utilizing the environmental score. Instead, the value one to one is adopted as the risk assessment. In many cases, this behavior leads to a dangerous evaluation which is incorrect for the overall system concerned.

CVSS Summary

With CVSS we have a system for evaluating and rating security gaps in

software. Since there are no alternatives, CVSS has become a defacto standard; the system has been in use worldwide for over ten years and is constantly being developed. The evaluation consists of three components.

First, there is the basic score, which is there to depict a purely technical worst-case scenario. The second component is the evaluation of the time-dependent corrections based on external influences, which includes further findings, tools, or patches for this security gap, which can be used to reduce the value. The third component of the assessment is your own system environment with regard to this vulnerability. With this consideration, the security gap is adjusted in relation to the real situation on site. Last but not least, an overall evaluation is made from these three values, which results in a number between 0.0 to 10.0.

This final value can be used to control your own organizational response to defend against the security gap. At first glance, everything feels quite abstract, so it takes some practice to get a feel for the application of CVSS, which can be developed through experience with your own systems.

Extent of Security Scanning

As soon as one deals with security, the following questions always come up: how much effort is enough, where should you start, and how quickly can you get the first results? In this chapter, we will deal with how to take these first steps. For this, we look first at two terms and consider the associated effects.

Time to Market

You have probably heard of the term “Time to Market,” but how does this relate to security? In general terms, this term means that the desired functionality is transferred as quickly as possible from conception through development into the production environment. This period should be as short as possible so that this functionality can be made available to the customer as quickly as possible. Following this optimization goal, all processing times are shortened as much as possible.

At first glance, it looks like a purely business-oriented perspective but on closer inspection turns out to be the same optimization target suitable for security-relevant aspects. Here, too, it is optimal to activate the required modifications to

the overall system as quickly as possible. In short, the term “time to market” is a common and worthwhile goal for security implementation.

Make or Buy

Across all layers of a cloud-native stack, the majority of the software and technology is bought or acquired rather than made. We will go through each of the layers in **Figure 6-1** and talk about the software composition.

Architecture diagram of a DevSecOps implementation

Figure 6-1. DevSecOps components that you can decide to build or purchase

The first step is the development of the application itself. Assuming that we are working with Java and using Maven as a dependency manager, we are most likely adding more lines of code indirectly as dependency compared to the number of lines we are writing by ourselves. The dependencies are the more prominent part, and third parties develop them. We have to be carefully, and it is good advice to check these external binaries for known vulnerabilities. We should have the same behavior regarding compliance and license usage.

The next layer will be the operating system, in our case Linux. And again, we are adding some configuration files and the rest are existing binaries. The result is an application running inside the operating system that is a composition of external binaries based on our configuration. The two following layers, Docker and Kubernetes, are leading us to the same result. Until now, we are not looking at the tool-stack for the production line itself. All programs and utilities that are directly or indirectly used under the hood called DevSecOps are some dependencies. All layers' dependencies are the most significant part by far. Checking these binaries against known vulnerabilities is the first logical step.

One-time and Recurring Efforts

Comparing the effort of scanning against known vulnerabilities and for compliance issues, we see a few differences. Let's start with the compliance issues.

Compliance Issues

The first step will be defining what licenses are allowed at what part of the production line. This definition of allowed license includes the dependencies during development and the usage of tools and runtime environments. Defining the non-critical license types should be checked by a specialized layer. With this list of white-labeled license types, we can start using the build automation to scan the full tool stack on a regular basis. After the machine has found a violation, we have to remove this element, and it must be replaced by another that is licensed under a white-labeled one.

Vulnerabilities

The ongoing effort on this side is low compared to the amount of work that vulnerabilities are producing. A slightly different workflow is needed for the handling of discovered vulnerabilities. With more significant preparations, the build automation can do the work on a regular basis as well. The identification of a vulnerability will trigger the workflow that includes human interaction. The vulnerability must be classified internally, which leads to the decision of what the following action will be.

How much is enough?

So let's come back to the initial question in this section. How much scanning is enough? Every change that has to do with adding or changing dependencies is a critical action in itself. It follows that each of these actions must lead to revalidation of the new status. Checking for known vulnerabilities or checking the license being used can be carried out very efficiently by automation. Another point that should not be underestimated is that the quality with which such an examination is carried out is constant, as nobody is involved at this point. If the value chain's speed is not slowed down by the constantly checking all dependencies, this is a worthwhile investment.

Compliance versus Vulnerabilities

Compliance Issues: Singular Points in your Full-Stack

There is one other difference between compliance issues and vulnerabilities. If there is a compliance issue, it is a singular point inside the overall environment. Just this single part is a defect and is not influencing other elements of the environment as shown in [Figure 6-2](#).

Circle diagram showing compliance issues in single layers of an application

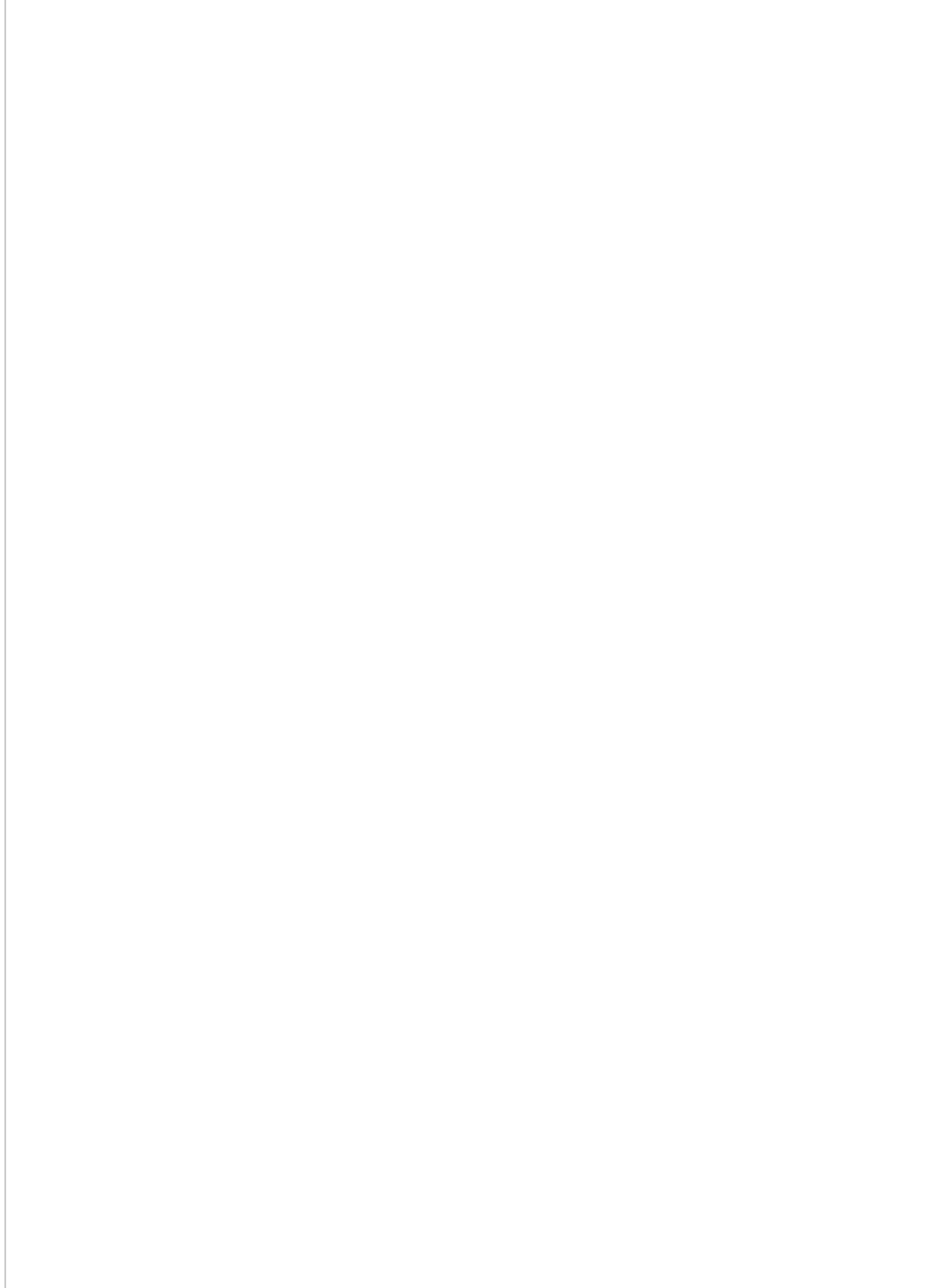


Figure 6-2. Layers of an application where compliance issues can be found

Vulnerabilities: Can be Combined into Different Attack-Vectors

Vulnerabilities are a bit different. They do not only exist at the point where they are located. Additionally, they can be combined with other existing vulnerabilities in any additional layer of the environment as shown in **Figure 6-3**. Vulnerabilities can be combined into different attack vectors. Every possible attack vector itself must be seen and evaluated. A set of minor vulnerabilities in different layers of the application can be combined into a highly critical risk.



Figure 6-3. Vulnerabilities in multiple layers of an application

Vulnerabilities: Timeline from Inception Through Production Fix

Again and again, we read something in the IT news about security gaps that have been exploited. The more severe the classification of this loophole, the more attention this information will get in the general press. Most of the time, one hears and reads nothing about all the security holes found that are not as well known as the SolarWinds Hack. The typical timeline of a vulnerability is shown in **Figure 6-4**.

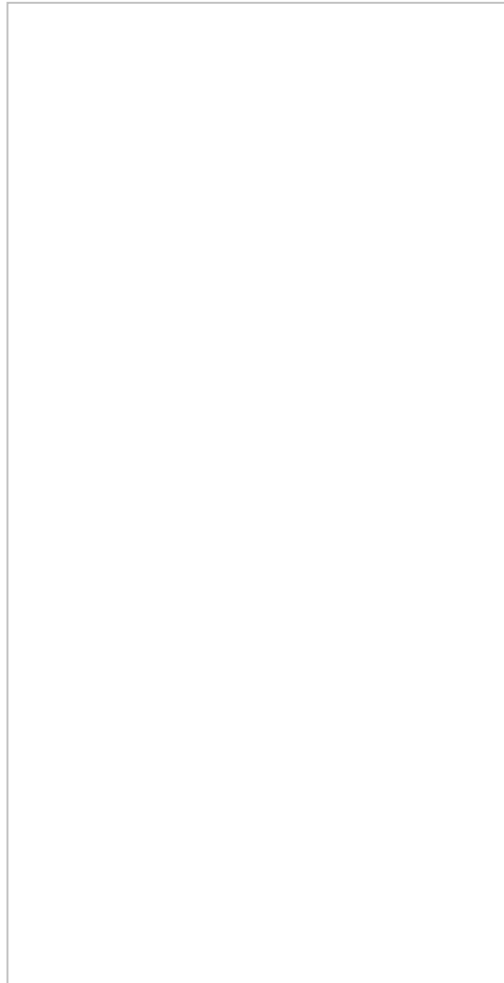


Figure 6-4. Timeline of a vulnerability

Creation of a Vulnerability

Let's start with the birth of a vulnerability. This can be done in two different ways. On the one hand, it can happen to any developer that has an unfortunate combination of source code pieces that creates a security hole. On the other hand, it can also be based on targeted manipulation. However, this has essentially no effect on the further course of the lifeline of a security vulnerability. In the following, we assume that a security hole has been created

and that it is now active in some software. These can be executable programs or libraries integrated into other software projects as a dependency.

Discovery of the Vulnerability

In most cases, it is not possible to understand precisely when a security hole was created. But let's assume that there is a security hole and that it will be found. It clearly depends on which person or which team finds this weak point. This has a serious impact on the subsequent course of the timeline. Let's start with the fact that this vulnerability has been found by people who have an interest in using this vulnerability themselves or having it exploited by other parties. Here the information is either kept under lock-and-key or offered for purchase at relevant places on the dark net. There are primarily financial or political motives here, which I do not want to go into here; however, at this point, it can be clearly seen that the information is passed on in channels that are not generally available to the public.

However, if the security gap is found by people or groups who are interested in making the knowledge about it available to the general public, various mechanisms now take effect. One must not forget that commercial interests will also be at play; however, the motivation is different. If it is the company or the project itself that is affected by this vulnerability, there is usually an interest in presenting the information as relatively harmless. The feared damage can even lead to the security gap being fixed, but knowledge about it further is hidden. This approach is dangerous, because one must assume that there will also be other groups or people who will gain this knowledge.

But let's assume that the information about the vulnerability was found by people who are not directly involved in the affected components. In most cases, this is motivated by profits from selling knowledge of the vulnerability. In addition to the affected projects or products, there are also vulnerability databases. These companies have a direct and obvious interest in acquiring this knowledge. But to which company will the finder of the vulnerability sell this knowledge? Here it can be assumed that it is very likely that it will be the company that pays the better price. This has another side effect that affects the classification of the vulnerability. Many vulnerabilities are given an assessment using the CVSS. Different people make the base value. Different people will have other personal interests here, which will then also be reflected in this value.

Here I refer again to the section on CVSS - Base Metric.

Regardless of the route via which the knowledge comes to the vulnerability databases, only when the information has reached one of these points can one assume that this knowledge will be available to the general public over time.

Public Availability of the Vulnerability

Regardless of which provider of vulnerabilities you choose, there will only ever be a subset of all known vulnerabilities in this data set. As an end-user, there is really only one way to go here. Instead of contacting the providers directly, you should rely on integrators. This refers to services that integrate a wide variety of sources themselves and then offer them processed and merged. It is also crucial that the findings are processed in such a way that further processing by machines is possible. This means that the meta-information such as the CVE or the CVSS value is included. This is the only way other programs can work with this information. The CVSS value is given as an example. This is used, for example, in CI environments to interrupt further processing when a specific threshold value is reached. Only when the information is prepared in this way and is available to the end-user can this information be consumed. Since the information generally represents a considerable financial value, it can be assumed in the vast majority of cases that the commercial providers of such data sets will have access to updated information more quickly than freely available data collections.

Fixing the Vulnerability in Production

When the information can now be consumed (i.e. processed with the tools used in software development) the storyline begins in your own projects. Regardless of which provider you have decided on, this information is available from a particular point in time, and you can now react to it yourself. The requirement is that the necessary changes are activated in production as quickly as possible. This is the only way to avoid the potential damage that can result from this security vulnerability. This results in various requirements for your own software development processes. The most obvious requirement is the throughput time. Only those who have implemented a high degree of automation can enable short response times in the delivery processes. It is also an advantage if the team concerned can easily make the necessary decisions themselves.

Lengthy approval processes are counterproductive at this point and can also cause extensive damage to the company.

Another point that can release high potential at this point is the provision of safety-critical information in all production stages involved. The earlier the information is taken into account in production, the lower the cost of removing security gaps. We'll come back to this in more detail when the shift-left topic is discussed.

Test Coverage is your Safety-Belt

The best knowledge of security gaps is of no use if this knowledge cannot be put to use. But what tools do you have in software development to take efficient action against known security gaps? I want to highlight one metric in particular: the test coverage of your own source code parts. If you have strong test coverage, you can make changes to the system and rely on the test suite. If a smooth test of all affected system components has taken place, nothing stands in the way of making the software available from a technical point of view.

But let's take a step back and take a closer look at the situation. In most cases, known security vulnerabilities are removed by changing the version used for the same dependency. This means that efficient version management gives you the agility you need to be able to react quickly. In very few cases, the affected components have to be replaced by semantic equivalents from other manufacturers. And in order to classify the new composition of versions of the same components as valid, strong test coverage is required. Manual tests would go far beyond the time frame and cannot be carried out with the same quality in every run. Mutation testing gives you a much more concrete test coverage than is usually the case with the conventional line or branch coverage.

To get a picture of the full impact graph based on all known vulnerabilities, it is crucial to understand all package managers included by the dependencies. Focusing on just one layer in the tech-stack is by far not enough. Package managers like Artifactory provide information, including the vendor-specific metadata. This can be augmented with security scanning tools like JFrog Xray that consume this knowledge and can scan all binaries that are hosted inside the repositories that are managed by your package manager.

Security scanning as a Promotion Quality Gate

Since the late 1990s, three generic success factors for IT projects have emerged that are independent of the project management method:

1. Participation and involvement of end-users as early as possible
2. The support of higher management
3. The formulation of clear business goals

The demand for comprehensive support from higher management provides, among other things, systematic control of the quality and project progress of IT projects in good time using criteria in order to be able to intervene. By specifying criteria, the management has two ways of controlling the software development process:

1. The criteria are project management specifications that the developer must adhere to. So the developers will do whatever it takes to meet these criteria.
2. The project management can intervene in the event of a deviation from the defined target. The target is determined by the criteria.

The name of the respective group can, of course, be different depending on the management system. The distribution of roles is also controversially discussed again and again. It turns out that a more substantial involvement of all team members leads to dynamic and successful structures.

In the context of project control, measures can be taken to counteract undesirable developments within a project. The ideal case for the project participants is the unrestricted continuation of the project. In extreme cases, however, it is also possible to cancel the project. Timeliness means being able to take action before significant financial damage can occur.

At the same time, however, this presupposes that relevant and measurable results are available to make effective project control sensible and possible. The end of activity within a project is a suitable time for this, as this is where results are available that can be checked. However, due to a large number of activities within a project, too frequent checks by the project management team would slow down the project progress. In addition, there would be a more significant

slow down the project progress. In addition, there would be a more significant burden on project management with many parallel projects (which would all have to be monitored).

A middle ground is to establish control and steering at specific significant points within the project as binding for each project. For this purpose, quality gates offer themselves as an opportunity to check the degree of fulfillment of the individual quality goals. A quality gate is a special point in time in a project at which a decision about the continuation or termination of a project is made based on a formal examination of quality-related criteria.

Metaphorically speaking, quality gates are barriers between the various process steps of a project: once the quality gate has been reached, a project can only be continued if all criteria, or at least a sufficiently large number of criteria, are met. This ensures that all results of the project at the time of the quality gate are good enough to be able to continue working with them. Using the criteria of a quality gate, the results on the one hand and the qualitative requirements for the results on the other can be determined. This means that they can be used to define the interfaces between individual project phases. To establish quality gates, certain structures, activities, roles, documents, and resources are necessary, which are summarized in a quality gate reference process.

The precise design of the Quality Gate reference process is based on the company's needs that want to establish Quality Gates. Quality gates have their origins in automobile development and in the production of technical goods, but they have increasingly found their way into system development projects and recently also into pure software development projects.

Quality gates in series production rely on statistically determined values that can be used as a target for control activities in future projects. Such a starting position does not exist in software development since software development projects are highly individual. As a result, a quality gate reference process practiced in assembly line production can only be transferred to software development to a limited extent. Instead, a suitable Quality Gate reference process must be designed differently in order to do justice to the particular problems of software development.

However, it makes sense to use the Quality Gate reference processes from other domains as they have been developed and optimized over the years. When using quality gates, two basic strategies have been identified. Depending on the

quality gates, two basic strategies have been identified. Depending on the objective, a company can choose one of these two strategies when designing a quality gate reference process:

Quality gates as uniform quality guideline:

Every project has to go through the same quality gates and is measured against the same criteria. The adaptation of a quality gate reference process that follows this strategy is (if at all) permissible to a minimal extent. The aim is to achieve at least the same level of quality in every project - a qualitative guideline is thus established for every project. Quality gates can therefore be used as a uniform measure of progress. The progress comparison between projects is possible in this way, as it can be checked which tasks have already passed a particular quality gate and which have not. The management can easily recognize when a project is behind another project (qualitatively) and act accordingly. Quality gates can thus easily be used as an instrument for multi-project management.

Quality gates as a flexible quality strategy:

The number, arrangement and selection of quality gates or criteria can be adapted to the needs of a project. Quality gates and standards can thus be tailored more precisely to the qualitative requirements of a project, improving the quality of results. However, this makes comparing between multiple projects more difficult. Fortunately, similar projects will have comparable quality gates and can be measured against similar criteria.

Dealing with the topic of Quality Gate through research on the Internet and in the literature (dissertations, standard works and conference volumes) provides a wide range of terms. On one hand, a synonymous term is used in many places; on the other hand, the concept is often equated with various other concepts. An example of the latter problem is equating the review concept or the milestone concept with the idea of the quality gate.

Fit with Project Management Procedures

The question arises whether this methodology can be applied to other project management processes. The answer here is a resounding YES. The quality gate methodology can be integrated into cyclical as well as a-cyclical project methods. The time sequence is irrelevant at this point and can therefore also be

used in classic waterfall projects at the milestone level.

The significant advantage is that this method can still be used in the case of a paradigm shift in project management. The knowledge built up in a team can continue to be used and do not lose their value. This means that the measures described here can be introduced and used regardless of the current project implementation.

Implementing Security with the Quality Gate Method

We will introduce, define and use a greatly simplified approach to integrate the cross-cutting issue of security.

In the following, we assume that the Quality Gate methodology is suitable for implementing any cross-sectional topic. The temporal component is also irrelevant and can therefore be used in any cyclical project management methodology. This approach is therefore ideally suited for integration into the DevSecOps project organization methodology.

The DevOps process is divided into different stages. The individual phases are seamlessly connected to one another. It makes no sense to install something at these points that interfere with the entire process. However, there are also much better places where cross-cutting issues are located. We are talking about the automated process derivation that can be found in a CI route. Assuming that the necessary process steps to go through a quality gate can be fully automated, a CI route is ideal for doing this regularly occurring work.

Assuming that the CI line carries out an automated process step, there are two results that can occur.

Green - Quality Gate has Passed

One possible result of this processing step is that all checks have passed successfully. Processing can therefore continue uninterrupted at this point. Only a few log entries are made to ensure complete documentation.

Red - Failed the Quality Gate

Another possible result may be that the check has found something that indicates a failure. This interrupts the process, and it must be decided what the cause of the failure was and how to remediate it. The automatic process usually ends at

the failure was and how to remediate it. The automatic process usually ends at this point and is replaced by a manual process. One question that arises at this point is that of responsibility.

Risk Management in Quality Gates

Since the quality gate is blocked by identifying a defect, it must be determined who is responsible for the following steps. A message that is triggered by the system requires an addressee.

The risk management that now takes place is comprised of the following activities:

- Risk assessment (identification, analysis, assessment and prioritization of risks)
- Design and initiation of countermeasures
- Tracking of risks in the course of the project

The risk determination was already completed with the creation of the criteria and their operationalization by weighing the requirements on a risk basis. This takes place at the final stage in the gate review itself.

The conception and initiation of countermeasures is an essential activity of a gate review, at least in the event that a project is not postponed or canceled. The countermeasures to be taken primarily counteract the risks that arise from criteria that are not met.

The countermeasures of risk management can be divided into preventive measures and emergency measures. The preventive measures include meeting the criteria as quickly as possible. If this is not possible, appropriate countermeasures must be designed. The design of the countermeasures is a creative act; it depends on the risk, its assessment and the possible alternatives. The effectiveness of the countermeasures must be tracked to ensure they are successful.

Practical Applications of Quality Management

Let's imagine we're about to release our software. For this purpose, all required components are generated and collected in the repository and every binary has

components are generated and consumed in the repository and every binary has an identity and version. All elements that are necessary for a release are put together in a deployment bundle after they have been created successfully. In this case, a release is a composition of different binaries in their respective version. The technology plays a subordinate role here, as the most diverse artifacts can come together in a release. You can also imagine that all crucial documents are part of this compilation at this point. This can mean documents such as the Release Notes and build information that provides information about the manufacturing process itself, for example, which JDK was used on which platform and much more. All information that can be automatically collated at this point increases the traceability and reproduction quality if a “post mortem” analysis has to be carried out. We now have everything together and would like to start making the artifacts available. We are talking about promoting the binaries here. This can be done in your own repositories or generally available global repositories. Now the last time has come when you can still make changes.

Now let’s take a few steps back and remember the SolarWinds hack only that we are now in place of SolarWinds. We are now in the process of releasing software and making it available to others. So it is high time to check whether all binaries we want to use correspond to what we have planned to provide.

We are talking about a security check as a promotional gateway. The tools used here should finally check two things. First, there are known vulnerabilities in the binaries that need to be removed. Second, all the licenses used in all the artifacts contained are adequate for the purpose. What becomes immediately clear here is the need for the check to be carried out independently of the technology used. This brings us back to the term “full impact graph.” At this point, we have to get the full impact graph in order to be able to achieve a high-quality result. The repository manager, who is responsible for providing all dependent artifacts, must be seamlessly integrated with the binary scanner. One example is the combination of Artifactory and Xray.

But is a security check a gateway for the promotion of binaries the earliest possible time? Where can you start earlier? We now come to the term “Shift Left.”

Shift Left to the CI and the IDE

Agile development, DevOps and the implementation of security have long been considered to be mutually exclusive. Classic development work was always confronted with the problem that the security of a software product could not be adequately defined as a final, static end state. The implementation does not seem to be specifiable in advance. This is the so-called security paradox in software development. The first attempts to define the security for a software product have led to the realization that security is a dynamic and continuous process and therefore cannot be carried out conclusively in the development of changing products.

At the same time, agile development seems too dynamic to be able to carry out a detailed security analysis of the software product to be developed in every development cycle. The opposite is the case because agile and secure development techniques complement each other very well. One of the key points of agile development is the ability to implement changes at short notice as well as changes to requirements within a short period of time.

In the past, security has tended to be viewed as a static process. Accordingly, agile implementation of these concepts from the security domain is required. The general handling of security requirements must adapt to this development in order to be able to be implemented efficiently. However, one must note that agile development is feature-oriented. With this approach, mine has a strongly function-driven development process. However, security requirements are mostly from the category of non-functional features and are therefore only available in an implicitly formulated form in most cases. The consequence of this and in combination with faulty security requirements engineering results in miscalculated development cycles with consequent increased time pressure; the sprint is canceled due to incorrect budget calculations, increased technical debts, persistent weak points or specific security gaps within the codebase. I am now primarily focusing on how the necessary conditions can be created in an agile development team that improves the code base's security level as early as possible. Regardless of the specific project management method used, the following approaches are not restricted in their validity.

It is essential to set the security level so that the respective development team should achieve a security increment when performing a product increment. A team with an implicit and pronounced security focus can immediately gain a

different level of security than a team without this focus. Regardless of the experience of each team, a general minimum standard must be defined and adhered to.

The OWASP Top 10 provides a list of general security vulnerabilities that developers can avoid with simple measures. Accordingly, they serve as an introduction to the topic and should be part of every developer's security repertoire. However, code reviews often reveal that teams are not adequately considering the "top 10," so this is a good area to focus teams on improvement.

It should also be recognized that developers can do an excellent job in their field but are not security experts. In addition to different levels of experience, developers and security experts have different approaches and ways of thinking that are decisive for their respective tasks. Therefore, the development team must be aware of their limitation with regard to the assessment of attack methods and security aspects. When developing critical components or in the event of problems, the organizational option of calling in a security expert must therefore be determined in advance. Nevertheless, developers should generally be able to evaluate typical security factors and take simple steps to improve the security of the code.

Ideally, each team has a member who has both development and detailed security knowledge. In the context of supported projects, the relevant employees are referred to as Security Managers (SecM). They monitor the security aspects of the developed code sections, define the so-called attack surface and attack vectors in each development cycle, support you in assessing the user stories' effort, and implement mitigation strategies.

In order to get a global overview of the codebase and its security level, it makes sense to aim for a regular exchange between the SecMs of the teams involved. Since a company-wide synchronization of the development cycle phases is unrealistic, the meetings of the SecMs should take place at regular, fixed times. In small companies or with synchronized sprints, the teams particularly benefit from an exchange during development cycle planning. In this way, cross-component security aspects and the effects of the development cycle on the security of the product increment can be assessed. The latter can currently only be achieved through downstream tests. Based on the development cycle review, a SecM meeting should also occur after implementing new components. In

conjunction, for the next sprint, the participants evaluate the security level

preparation for the next sprint, the participants evaluate the security level according to the increment. There is conceptual overlap with the OWASP Security Champion.

However, a key difference is that a SecM is a full-fledged security expert with development experience who acts on the same level as the senior developer. The OWASP Security Champions are implemented differently, but these are often developers, possibly junior developers, who acquire additional security knowledge that can be very domain-specific depending on experience. When implementing secure software, however, it is crucial to take into account the security-relevant effects of implementation decisions and cross-thematic specialist knowledge.

Regardless of whether a team can create a dedicated role, there are some basic measures to support the process of developing secure software. These are the following best practice recommendations and empirical values.

Not All Clean Code is Secure Code

The book “Clean Code” by Robert Martin, also known as Uncle Bob, coined the term “Clean Code”. However, a common misconception among decision-makers is that the correct implementation of a feature implies clean code while also covering the security of the code.

Safe and clean code overlap but are not the same. “Clean code” promotes understandability, maintainability, and reusability of code. Secure code, on the other hand, also requires its predefined specifications and compliance with them. However, clean code is often a requirement for safe code. The code can be written cleanly without any security features. However, only a clean implementation opens up the full potential for security measures.

Well-written code is also better to secure because the relationships between components and functions are clearly defined and delimited. Any development team looking for reasons to promote adherence to and implementation of the “Clean Code” principles will find good arguments in the security of the code, which can also be explained economically to decision-makers.

Effects on Scheduling

In general, and particularly in agile development, teams do not allow enough

in general, and particularly in agile development, teams do not allow enough time to improve the code base when planning the next version. In sprint planning, the focus on effort assessment is primarily on time to develop a new function. Hardening is only considered explicitly when there is a special requirement.

How much time teams need to implement a function safely depends on the functionality, the status of the product increment, the existing technical debt and the prior knowledge of the developer. However, as intended in agile development, it should be up to the team to estimate the actual time required. Since miscalculations are to be expected, especially at the beginning, it can make sense to reduce the number of user stories adopted compared to the previous sprints.

The Right Contact Person

Every team must have access to security professionals. IT security is divided into numerous, sometimes highly specific and complex sub-areas for which full-time security experts are responsible. Good programmers are full-time developers and don't have an entire focus on security. After IT security training, developers cannot replace IT, security experts.

It is the responsibility of project management to ensure that organizational, structural, and financial requirements are met so that teams can quickly draw on technical expertise when needed and during an assessment. This is not the case by default in most units.

Dealing with Technical Debt

Alternatively, the sub-strategy is to set a fixed portion of the estimated project time for servicing technical debt. The approach is minor as there is a risk that teams will use the time spent processing technical debt to implement a function instead and misjudge the extent of technical debt under the development cycle pressure.

Technical debt is an integral part of development, and project owners should treat it as such - both in terms of time and budget. Technical debt has a negative impact on the maintainability, development, and security of the codebase. This means a significant increase in individual (new) implementation costs and a

sustained slowdown in overall production by blocking developers for a more extended period of time with individual projects. Therefore, it is in the interest of everyone involved - especially management - to keep the technical debt of a codebase low and continuously reducing it.

Advanced Training on Secure Coding

Typically, there is a list of secure coding guidelines in a public folder somewhere. In addition, the OWASP Top 10 are often published in to the general public. The fallacious conclusion is that safety can be learned in passing and everyone has access to the necessary materials. As a rule, however, employees do not read such documents or, at best, skim them. Often, after a while, teams no longer know where such documents are, let alone what use they should get from them. Admonitions to encourage reading the guidelines are not very helpful if companies cannot create extra time to focus on secure coding.

Milestones for Quality

Quality gates in development help to check compliance with quality requirements. Analogous to the so-called Definition of Done (DoD), the team-wide definition of when a task can be viewed as completed, quality gates should not only be available in stationary paper form. Ideally, automated checks are integrated into the CI / CD pipeline through static code analyses (SAST, Static Application Security Testing) or the evaluation of all dependencies.

For developers, however, it can be helpful to receive feedback on the code and its dependencies in addition to server-side feedback during programming. There are some language- and platform-dependent IDE plug-ins and separate code analysis tools such as Findbugs / Spotbugs, Checkstyles or PMD. When using JFrog Xray, the IDE plug-in can be used to make it easier to compare against known vulnerabilities and compliance issues. An additional upstream process for checking the code in the IDE pursues the goal of familiarizing developers with security aspects during development. The code security is improved not only at the points identified by the plug-in, but in the entire code, since developers are given a security orientation. Another side effect is the reduction in the number of false positives on the build server. The latter is particularly high for security quality gates, as security gaps in the code are often context-dependent and

require manual verification, which leads to a huge increase in the development effort.

The Attacker's Point of View

Evil user stories (also called bad user stories) are derived from the technical user stories and serve to present the desired functionality from an attacker's perspective. Analogous to user stories, they are designed so that their focus is not on the technical implementation. Accordingly, people with a small technical background in IT security can write user stories. However, this increases the effort required to generate tasks from the possibly unspecific (bad) user stories.

Ideally, bad user stories try to depict the attack surface. They enable the development team to process identified attack methods in a familiar workflow. This creates awareness of possible attack vectors, but these are limited. Evil user stories are, on the one hand, limited by the knowledge and experience of their respective authors and their imagination; on the other hand, through the developer's ability to fend off the attack vector in the context of the sprint. It's not just about whether the developers develop the right damage control strategy but also about correctly and comprehensively identifying the use case in the code.

Like conventional user stories, the evil variants are not always easy to write. Teams with little experience in developing secure software, in particular, can encounter difficulties creating meaningful nasty user stories. If there is a SecM on the team, it should essentially take on the task or offer support. Teams without SecM should either look for external technical expertise or plan a structured process for creating the evil User Stories. The OWASP Cornucopia Cards shown below can help.

Methods of Evaluation

In order to establish security as a process within agile development, regular code reviews must be carried out, with the focus on the security level of the code, both component-by-component and across segments. Ideally, errors that are easy to avoid and can cause security breaches can be identified and corrected as part of the CI/CD pipeline through quality gates and automated tests. In this case, the component-by-component test is primarily concerned with the investigation of

the attack surface of the respective component and the weakening of attack vectors. A cheat sheet for analyzing the attack surface can be found on the OWASP Cheat Sheet Series GitHub.

The teams must regularly redefine the attack surface, as it can change with each development cycle. The cross-component check is used to monitor the attack surface of the overall product, as it can also change with every development cycle. Ultimately, only a cross-component view enables the search for attack vectors that result from interactions between components or even dependencies.

If SecMs are not available, a security assessment can be carried out through a structured approach and joint training in the team. The OWASP Cornucopia can, among other things, promote such an approach. The Cornucopia is a card game that is freely available on the project website. The players try to apply the attack scenarios depicted on the cards to a domain selected in advance by the team or, if necessary, only to individual methods, such as the codebase. The team must then decide whether the attack scenario of the card played is conceivable. Therefore the focus is on the identification of attack vectors; Due to time constraints, mitigation strategies should be discussed elsewhere. The winner of the card game is the one who could successfully play the most difficult cards. The team must document the resulting security analysis at the end.

One benefit of Cornucopia is that it increases awareness of code vulnerabilities across the team. The game also improves the developer's expertise in IT security. The focus is on the ability of the developer and thus reflects agile guidelines. Cornucopia sessions are an excellent tool to generate evil user stories afterwards.

The problem with the Cornucopia sessions is that they present especially inexperienced teams with a steep learning curve at the beginning. There is also a risk that the team will incorrectly discard a potential attack vector. If the preparation is poor (e.g. components that are too large or not enough technical knowledge about possible attack vectors) Cornucopia can be inefficient in terms of time. Therefore, it is advisable, especially in the first few sessions, to examine small, independent components and, if necessary, to consult security experts.

Be Aware of Responsibility

Overall it can be said that developers should not allow the code security scanner

Overall, it can be said that developers should not allow the code security aspects to be taken out of their hands. Ideally, the team should jointly insist that there is sufficient time and financial resources to implement basic security aspects.

Current developers will largely define and shape the world for the years to come. Due to the expected digitization and networking, security must not fall victim to budget and time constraints. According to the agile manifesto, the codebase remains the product of the team responsible for the outcome.

Chapter 7. Mobile Workflows

Stephen Chin

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors’ raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 9th chapter of the final book. If there is a GitHub repo associated with the book, it will be made active after final publication.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at ccollins@oreilly.com.

Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence.

—Edsger Dijkstra

Coverage of DevOps wouldn’t be complete without talking about mobile development and smartphones, which is the fastest growing segment of computer ownership. The past decade has seen a meteoric rise in smartphone usage, with billions of smartphones owned globally, as shown in [Figure 7-1](#).



*Figure 7-1. Number of smartphone users in the world from 2012 through 2023 according to [Statista](#) (prediction for 2023 marked with *)*

Smartphone ownership is expected to continue to rise since many large countries such as India and China have less than 70% ownership. With over 3.6 billion smartphones in the world today and an expected 4.3 billion smartphones by the year 2023, this is a market and user base that can’t be ignored.

Smartphones also have another property that makes DevOps an essential practice, which is that they fall into a class of internet connected devices where continuous updates are expected by default because they are targeted at

continuous updates are expected by default, because they are targeted at consumers who are less technical and need to maintain their devices with minimal user involvement. This has been propelled by the app ecosystem built around smartphones, which makes downloading new software and receiving software updates easy and relatively low risk for end users.

There are several functional reasons you may want to update your app:

- *Adding new features for users:* Most apps are released quickly and with a minimum viable set of features to reduce time to market. This allows for frequent small feature updates to add useful functionality for end users.
- *Fixing bugs and improving the stability of your application:* More mature applications have lots of updates that fix small bugs, stability issues, and user experience improvements. These changes are typically minor and can be released at a very frequent cadence.
- *Patching security vulnerabilities or exploits:* Mobile applications typically have a very large attack surface which includes the locally installed app, the backend that supplies data, and user authentication workflows for app and cloud service logins.

In addition to those reasons, a lot of app updates are driven by the need to increase market share and improve engagement with users. Some examples of updates that help to grow market share of your app include the following:

- *Aligning with major platform releases* - Whenever a major platform release occurs, apps that are certified against the new version and updated to take advantage of new functionality will see an increase in downloads.
- *Raising visibility of your app in the store* - App stores reward apps that frequently update by retaining user ratings between releases and highlighting new releases. The release notes also gives you a chance to increase searchable content in the store. In contrast, if your app stagnates with no updates, then it will naturally drop in search engine optimization.
- *Reminding current users about the existence of your application to*

increase utilization - Mobile platforms prompt users about updates to their existing apps and sometimes display badges or other reminders that will increase engagement.

The top applications in the app stores know the importance of continuous updates and update very frequently. According to [appbot](#), of the 200 top free apps, the median time since the last update was 7.8 days! With this pace of updates, if you do not make use of a continuous release process, you won't be able to keep up.

Java developers have some great options for building mobile applications. These include mobile-focused web development with responsive web apps that adapt to constrained devices. Also dedicated mobile applications written in Java for Android devices. And finally several cross-platform options for building applications that work across Android and iOS devices, including Gluon Mobile and Electron.

This chapter focuses primarily on Android application development; however, all the same mobile DevOps techniques and considerations apply across these Java-based mobile platforms.

Fast-paced DevOps workflows for mobile

Here are some of the business benefits you will realize from investing in mobile DevOps:

- *Better customer experience:* With the easy and accessible rating systems available in the app store, customer experience is king. By being able to respond quickly to customer issues and test on large variety of devices, you will ensure an optimal customer experience.
- *Faster innovation:* By continuously releasing to production you will be able to get new features and capabilities to your customers at a higher velocity than your competitors.
- *Higher software quality:* With the large number and high fragmentation of Android devices, it is impossible to thoroughly test your application manually. But with an automated mobile testing strategy that hits the

key device characteristics of your user base, you will reduce the number of issues reported by end users.

- *Reduced risk*: The majority of executable code in modern applications has open source dependencies that expose you to known security vulnerabilities. By having a mobile DevOps pipeline that allows you to test new versions of dependencies and update frequently, you will be able to quickly fix any known vulnerabilities in your application before they can be taken advantage of.

The same principles and best practices outlined in the rest of this book apply for mobile application development, but are amplified 10x by the size and expectations of this market. When planning out a mobile DevOps pipeline for Android devices, here are the stages you need to consider:

- Build - Android build scripts are usually written in Gradle. As a result, you can use any continuous integration server of your choice, including Jenkins, CircleCI, Travis CI, or JFrog Pipelines.
- Test
 - *Unit tests*: Android unit tests are typically written in JUnit, which can easily be automated. Higher level Android unit tests are often written in some sort of UI test framework like Espresso, Appium, Calabash, or Robotium.
 - *Integration tests*: Besides testing your own application, it is important to test interactions between applications with tools like UI Automator that are focused on integration testing and can test across multiple Android applications.
 - *Functional tests*: Overall application verification is important. You can do this manually, but automated tools can simulate user input like the previously mentioned UI automation tools. Another option is to run robotic crawler tools like Google's App Crawler in order to inspect your application's user interface and automatically issue user actions.
- Package - In the package step you aggregate all of the scripts,

configuration files, and binaries needed for deployment. By using a package management tool like Artifactory you retain all the build and test information and can easily trace dependencies for traceability and debugging.

- Release - One of the best parts of mobile app development is that releasing mobile applications ends with the app store submission and the final deployment to devices is managed by the Google Play infrastructure. The challenging parts are that you have to prepare your build to make sure the app store submission is successful, and you'll be penalized for any mistakes in building, testing, and packaging by delays if you do not fully automate this.

As you can see, the biggest difference in DevOps for Android development comes with testing. There is a lot of investment in UI test frameworks for Android apps, because automated testing is the only solution to the problem of testing across a highly fragmented device ecosystem. We will find out exactly how severe the Android device fragmentation is in the next section, and talk about ways to mitigate this later in the chapter.

Android Device Fragmentation

The iOS ecosystem is tightly controlled by Apple, which limits the number of hardware models available, the variations in screen size, and the set of hardware sensors and features on their phones. Since 2007 when the first iPhone debuted only 29 different devices have been produced, only 7 of which are currently sold.

In contrast, the Android ecosystem is open to a plethora of different device manufacturers who customize everything from the screen size and resolution to the processor and hardware sensors, and even produce unique form factors like foldable screens. There are over 24,000 different devices from 1,300 different manufacturers, which is 1000x more fragmentation than iOS devices. This makes testing for Android platforms much more difficult to execute on.

When it comes to fragmentation, several key differences make it hard to uniformly test different Android devices:

- **Android Version** - Android device manufacturers do not always provide updates for older devices to the latest Android version, creating a situation where users are stuck on old Android OS versions until they buy a new device. As shown in **Figure 7-2** the dropoff in use of old Android versions is very gradual, with active devices still running 7+ year old Android 4.x releases, including Jelly Bean and KitKat.
- **Screen Size and Resolution** - Android devices come in a wide array of different form factors and hardware configurations with a trend towards larger and more pixel dense displays. A well designed application needs to scale to work well across a range of different screen sizes and resolutions.
- **3D Support** - Particularly for games, it is critical to know what level of 3D support you will get on devices both in terms of APIs and also performance.
- **Hardware Features** - Most Android devices come with the basic hardware sensors (camera, accelerometer, GPS), but there is a lot of variation for support of newer hardware APIs like NFC, barometers, magnetometers, proximity and pressure sensors, thermometers, and so on.

Layered graph showing the adoption and decline of major Android releases

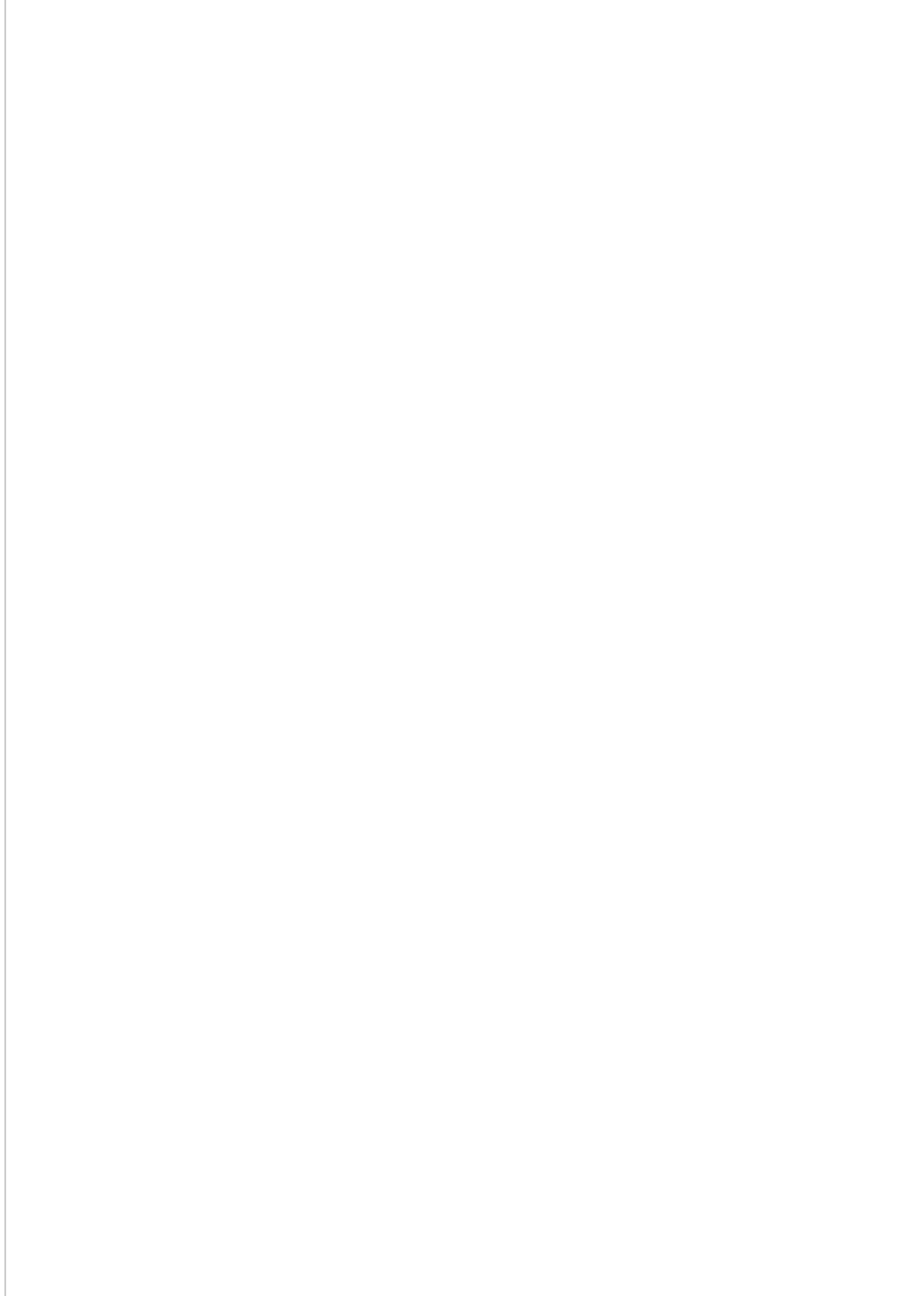


Figure 7-2. Percentage of devices utilizing major Android versions from 2010 through 2020¹

Android OS Fragmentation

Android version fragmentation affects device testing at two different levels. The first is the major Android version, which determines how many different Android API versions you need to build for and also test against. And the second is the OS customization done by original equipment manufacturers (OEMs) to support specific hardware configurations.

In the case of iOS, since Apple controls the hardware and the operating system, it is able to push out updates for all supported devices simultaneously. This keeps the adoption level of minor updates for performance and security fixes very high. Apple also puts a lot of features and marketing into major releases to push the installed base to upgrade to the latest version quickly. As a result, Apple was able to achieve **86% adoption** of iOS 14 only seven months after its initial release.

The Android market is significantly more complex since OEMs modify and test custom versions of Android OS for their devices. Also, they are reliant on System on Chip (SoC) manufacturers to provide code updates for different hardware components. This means that devices created by major vendors are likely to receive only a couple major OS version updates, and devices from smaller vendors may never see an OS upgrade even when they are under support.

To help you decide how far back you should support different Android OS versions, Google provides information in Android Studio on the device adoption by API level. The distribution of users as of August 2021 is shown in **Figure 7-3**. To achieve > 86% adoption comparable to the latest iOS version you need to support at least Android 5.1 Lollipop, which is a release that came out in 2014. Even then you are still missing out on over 5% of users who are still using Android 4 based devices.

List of Android versions with information on usage and features

Figure 7-3. Screenshot of Android Studio showing the distribution of users on different versions of the Android platform (Android 11 has < 1% adoption)

To further complicate the situation, every OEM modifies the Android OS it ships for its devices, so it is not enough to simply test one device per major Android version. This is a result of the way Android uses the Linux kernel to access hardware devices.

The Linux kernel is the heart of the operating system, and provides the low-level device driver code to access cameras, accelerometers, the display, and other hardware on the device. To the Linux kernel that Android is based on, Google adds in Android specific features and patches, SoC vendors add in hardware specific support, and OEMs further modify it for their specific devices. This means that each device has a range of variation in performance, security, and potential bugs that could affect your application when a user runs it on a new device.

Google worked towards improving this situation with Android 8.0 Oreo, which includes a new hardware abstraction layer that allows device-specific code to run outside the kernel. This allows OEMs to update to new Android kernel versions from Google without waiting for device driver updates from SoC vendors, which reduces the amount of redevelopment and testing required for OS upgrades. However, other than Pixel devices that Google handles OS updates for, the majority of Android device upgrades are in the hands of OEMs, which are still slow to upgrade to new Android versions.

Building for Disparate Screens

Given the diversity in hardware manufactures and over 24,000 models, as discussed in the previous section, it should be no surprise that there is also a huge variation in screen sizes and resolutions. New screen dimensions are constantly being introduced, such as the enormous HP Slate 21, which uses a 21.5 inch touchscreen, and the Galaxy Fold with a vertical 1680x720 cover display that opens to reveal a double-wide inner display with a resolution of 2152x1536.

Besides the huge variation in screen sizes, there is a constant battle over achieving higher pixel density as well. Higher pixel densities allow for clearer text and sharper graphics, providing a better viewing experience.

The current front runner in pixel density is the Sony Xperia XZ, which packs a 3840x2160 USH-1 display in a screen that measures only 5.2 inches diagonally. This gives a density of 806.93 PPI, which is getting close to the maximum resolution the human eye can distinguish.

Applied Materials, one of the leading manufacturers of LCD and OLED displays, did research on human perception of pixel density on handheld displays. They found that at a distance of 4 inches from the eye a human with 20/20 vision can distinguish 876 PPI². This means that smartphone displays are quickly approaching the theoretical limit on pixel density; however, other form factors like virtual reality headsets may drive the density even further.

To handle variation in pixel densities, Android categorizes screens into the following pixel density ranges:

- *ldpi* - ~120dpi (.75x scale): Used on a limited number of very low resolution devices like the HTC Tattoo, Motorola Flipout, and Sony X10 Mini, all of which have a screen resolution of 240x320 pixels.
- *mdpi*: ~160dpi (1x scale): This is the original screen resolution for Android devices such as the HTC Hero and Motorola Droid
- *tvdpi* - ~213dpi (1.33x scale): Resolution intended for televisions such as the Google Nexus 7, but not considered a “primary” density group
- *hdpi* - ~240dpi (1.5x scale): The second generation of phones such as the HTC Nexus One and Samsung Galaxy Ace increased resolution by 50%
- *xhdpi* - ~320dpi (2x scale): One of the first phones to use this 2x resolution was the Sony Xperia S, followed by phones like the Samsung Galaxy S III and HTC One
- *xxhdpi* - ~480dpi (3x scale): The first xxhdpi device was the Nexus 10 by Google, which was only 300dpi but needed large icons since it was in tablet form factor
- *xxxhdpi* - ~640dpi (4x scale): This is currently the highest resolution used by devices like the Nexus 6 and Samsung S6 Edge

As displays continue to increase in pixel density, Google will likely introduce a fifth

As displays continue to increase in pixel density, Google probably wishes it had chosen a better convention for high resolution displays than just adding more “x”s!

To give the best user experience for your end users, it is important to have your application look and behave consistently across the full range of different available resolutions. Given the wide variety of different screen resolutions it is not enough to simply hard code your application for each resolution.

Here are some best practices to make sure that your application will work across the full range of resolutions:

- Always use density independent and scalable pixels
 - Density independent pixels (dp) - Pixel unit that adjusts based on the resolution of the device. For an mdpi screen 1 pixel (px) = 1 dp. For other screen resolutions $px = dp * (dpi / 160)$.
 - Scalable pixels (sp) - Scalable pixel unit used for text or other user resizable elements. This starts at 1 sp = 1 dp and adjusts based on the user defined text zoom value.
- Provide alternate bitmaps for all available resolutions
 - Android allows you to provide alternate bitmaps for different resolutions by putting them in subfolders named “drawable-?” where “?dpi” is one of the supported density ranges.
 - The same applies for your app icon, except you should use subfolders named “mipmap-?” so that the resources are not removed when you build density-specific APKs, because app icons are often upscaled beyond the device resolution.
- Better yet, use vector graphics whenever possible
 - Android Studio provides a tool called “Vector Asset Studio” that allows you to convert an SVG or PSD into an Android Vector file that can be used as resource in your application as shown in **Figure 7-4**.



Figure 7-4. Conversion of an SVG file to an Android Vector format resource

Building applications that cleanly scale to different screen sizes and resolutions is complicated to get right and therefore needs to be tested on different

resolution devices. To help out with focusing your testing efforts, Google provides user-mined **data** on the usage of different device resolutions as shown in **Table 7-1**.

Table 7-1. Android screen size and density distribution

| | ldpi | mdpi | tvdpi | hdpi |
|--------|------|------|-------|-------|
| Small | 0.1% | | | |
| Normal | | 0.3% | 0.3% | 14.8% |
| Large | | 1.7% | 2.2% | 0.8% |
| Xlarge | | 4.2% | 0.2% | 2.3% |
| Total | 0.1% | 6.2% | 2.7% | 17.9% |

As you can see, some resolutions are not very prevalent and, unless your application targets these users or legacy device types, you can prune them from your device-testing matrix. LDPI is used on only a small segment of Android devices and with only 0.1% market share — few applications are optimized for this very small resolution screen. Also, tvdpi is a niche screen resolution with only 2.7% usage and can be safely ignored since Android will automatically downscale hdpi assets to fit this screen resolution.

This still leaves you with five different device densities to support and a potentially innumerable number of screen resolutions and aspect ratios to test. I discuss testing strategies later, but you will likely be using some mix of emulated devices and physical devices to make sure that you provide the best user experience across the very fragmented Android ecosystem.

Hardware and 3D Support

The very first Android device was the HTC Dream (aka T-Mobile G1) shown in **Figure 7-5**. It had a medium density touchscreen of 320x480px, a hardware keyboard, speaker, microphone, five buttons, a clickable trackball, and a rear-mounted camera. While primitive by modern smartphone standards, it was a great platform to launch Android, which lacked support for software keyboards

at the time.

Figure 7-5. The T-Mobile G1 (aka. HTC Dream), which was the *first smartphone* to run the Android operating system. Photo used under *Creative Commons license*.

By comparison with modern smartphone standards this was a very modest hardware set. The Qualcomm MSM7201A processor that drove the HTC Dream was a 528 MHz ARM11 processor with support for only OpenGL ES 1.1. In comparison, the Samsung Galaxy S21 Ultra 5G sports a 3200 x 1440 resolution screen with the following sensors:

- 2.9 GHz 8-core processor
- ARM Mali-G78 MP14 GPU with support for Vulkan 1.1, OpenGL ES 3.2, and OpenCL 2.0
- Five cameras (1 front, 4 rear)
- Three microphones (1 bottom, 2 top)
- Stereo speakers
- Ultrasonic fingerprint reader
- Accelerometer
- Barometer
- Gyro sensor (gyroscope)
- Geomagnetic sensor (magnetometer)
- Hall sensor
- Proximity sensor
- Ambient light sensor
- Near-field communication (NFC)

The flagship Samsung phones are at the high end of the spectrum when it comes to hardware support, and include almost all of the supported sensor types. Phones meant for mass market may choose to use less powerful chipsets and leave off sensors to reduce cost. Android uses the data from the available physical sensors to also create some “virtual” sensors in software that are used by applications:

- *Game rotation vector*: Combination of data from the accelerometer and gyroscope
- *Gravity*: Combination of data from the accelerometer and gyroscope (or magnetometer if no gyroscope is present)
- *Geomagnetic rotational vector*: Combination of data from the accelerometer and magnetometer
- *Linear acceleration*: Combination of data from the accelerometer and gyroscope (or magnetometer if no gyroscope is present)
- *Rotation vector*: Combination of data from the accelerometer, magnetometer, and gyroscope
- *Significant motion*: Data from the accelerometer (and possibly substitutes other sensor data when in low power mode)
- *Step detector/counter*: Data from the accelerometer (and possibly substitutes other sensor data when in low power mode)

These virtual sensors are only available if a sufficient set of physical sensors are present. Most phones contain an accelerometer, but may choose to omit either a gyroscope or magnetometer or both, reducing the precision of motion detection and disabling certain virtual sensors.

Hardware sensors can be emulated, but it is much harder to simulate real world conditions for testing. Also, there is much more variation in hardware chipset and SoC vendor driver implementation, producing a huge test matrix required to verify your application across a range of devices.

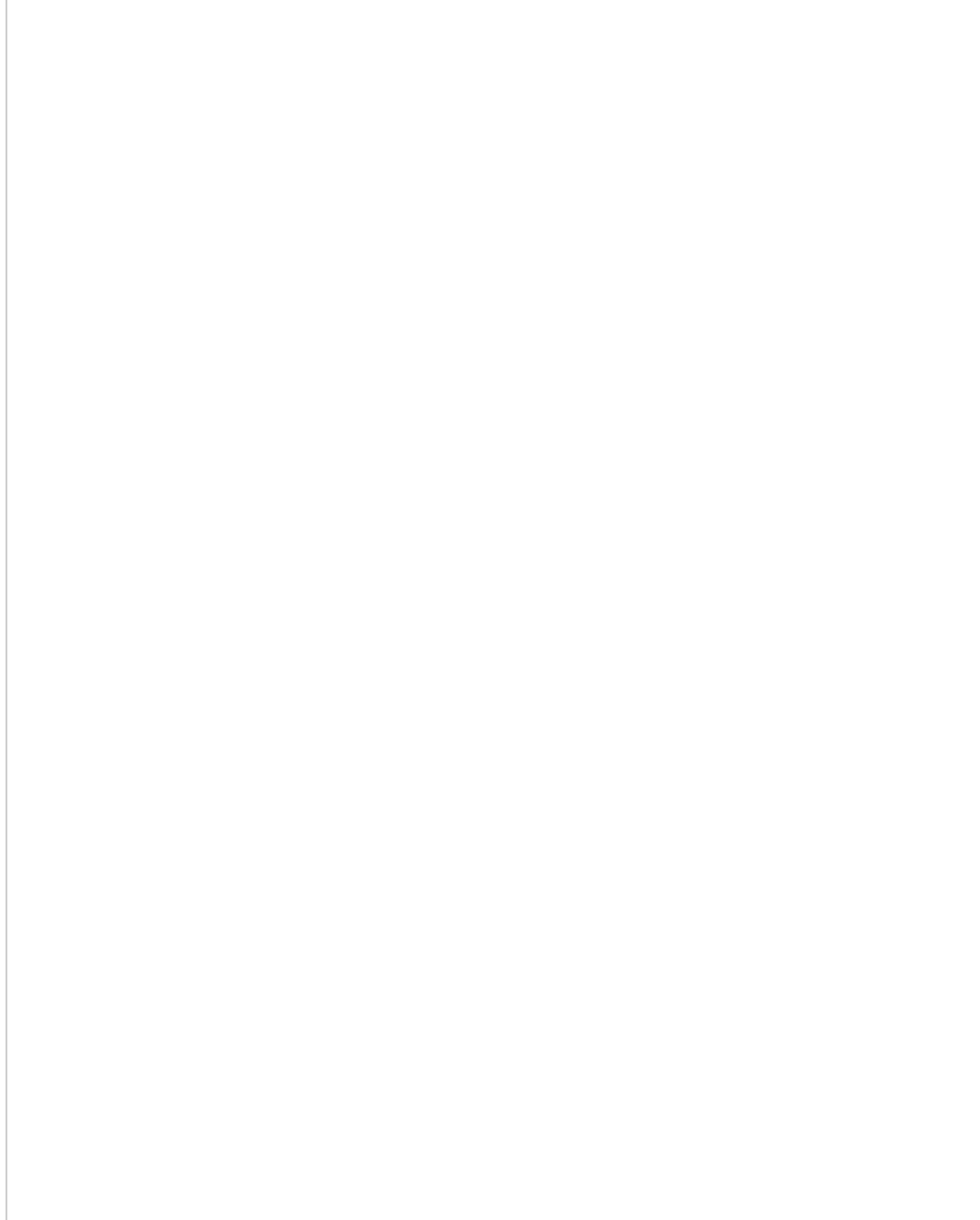
The other aspect of hardware that is particularly important for game developers, but increasingly is part of the basic graphics stack and expected performance of applications, is 3D API support. Almost all mobile processors support some basic 3D APIs, including the first Android phone, which had support for OpenGL ES 1.1, a mobile-specific version of the OpenGL 3D standard. Modern phones support later versions of the OpenGL ES standard including OpenGL ES 2.0, 3.0, 3.1, and now 3.2.

OpenGL ES 2.0 introduced a dramatic shift in the programming model,

switching from a functional pipeline to a programmable pipeline, allowing for more direct control to create complex effects through the use of shaders. OpenGL ES 3.0 further increased the performance and hardware independence of 3D graphics by supporting features like vertex array objects, instanced rendering, and device independent compression formats (ETC2/EAC).

OpenGL ES adoption has been rather quick with all modern devices supporting at least OpenGL ES 2.0. According to Google's device data shown in **Figure 7-6**, the majority of devices (67.54%) support OpenGL ES 3.2, the latest version of the standard released on August 2015.

Pie chart showing of OpenGL ES version adoption



*Figure 7-6. Percentage of Android devices adopting different versions of OpenGL ES from **Google's** **Distribution Dashboard**.*

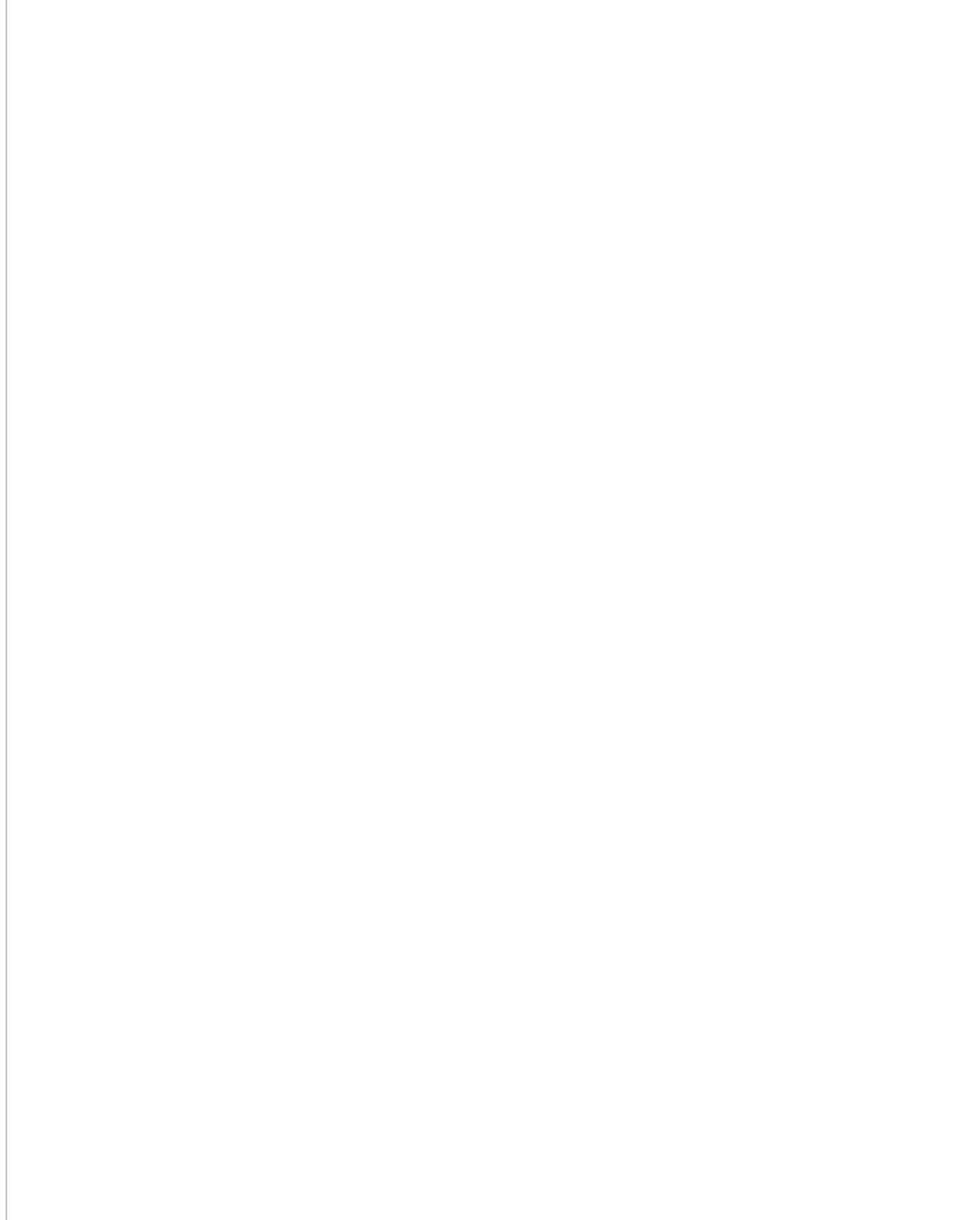
Vulkan is a newer graphics API that modern graphics chipsets support. It has the advantage of being portable between desktop and mobile allowing for easier

porting of desktop code as computing platforms continue to converge. Also, it allows an even finer level of control over threads and memory management, and an asynchronous API for buffering and sequencing commands across multiple threads, making better use of multi-core processors and high-end hardware.

Since Vulkan is a newer API, adoption has not been as quick as OpenGL ES; however, 64% of Android devices have some level of Vulkan support.

According to Google's device statistics visualized in **Figure 7-7** this is split between Vulkan 1.1, which is supported by 42% of devices, and the remaining 22% of devices that only support the Vulkan 1.0.3 API level.

Pie chart showing Vulkan version adoption



*Figure 7-7. Percentage of Android devices adopting different versions of Vulkan from **Google's** Distribution Dashboard.*

Similar to hardware sensor testing, there are a large variety of 3D chipsets implemented by different manufacturers. Therefore, the only way to reliably test

for bugs and performance issues in your application is to execute on device testing of different phone models, covered in the next section.

Continuous Testing on Parallel Devices

The last section discussed the large amount of fragmentation in the Android device ecosystem. This is forced both by technological factors like the Android OS architecture as well as the complex ecosystem of OEMs and SoC vendors. Also, the sheer popularity of the Android platform, with 1,300 manufacturers producing over 24,000 devices, creates a continuous testing and deployment challenge.

Device emulators are great for development and basic testing of applications, but cannot possibly simulate the complex interactions of unique hardware configurations, device drivers, custom kernels, and real-world sensor behavior. Therefore, a high level of manual and automated testing on devices is required to ensure a good experience for end users.

There are two basic approaches to doing hardware testing at scale. The first is to build out your own device lab with shared devices. This is a practical approach to get started with testing since you likely have a large collection of Android devices available that could be put to better use with proper infrastructure and automation. However, depending upon the number of device configurations you want to support this can be quite a large and expensive endeavor. Also, the ongoing maintenance and upkeep for a large device farm can be quite costly both in materials and labor.

The second option is to outsource your device testing to a cloud service. Given the advances in remote control of Android devices and stability of the platform, it is very convenient to be able to select your matrix of devices and have your automated tests fired off in the cloud. Most cloud services offer detailed screenshots and diagnostic logs that can be used to trace build failures and also the ability to manually control a device for debugging purposes.

Building a Device Farm

Building your own device farm, even at a small scale, is a great way to leverage Android devices that you already have and increase their utility for your entire

Android devices that you already have and increase their utility for your entire organization. At scale, device farms can significantly reduce the run rate cost of Android development once you have made the upfront investment in hardware. Keep in mind, though, running a large device lab is a full time job and has ongoing costs that need to be accounted for.

A popular open source library for managing Android devices is Device Farmer (formerly Open STF). Device Farmer allows you to remote control an Android device from your web browser with a real-time view of the device screen, as shown in **Figure 7-8**. For manual tests you can type from your desktop keyboard and use your mouse to input single or multitouch gestures. For automated tests there is a REST API that allows you to use test automation frameworks like Appium.

Picture of a volunteer built device lab at a conference

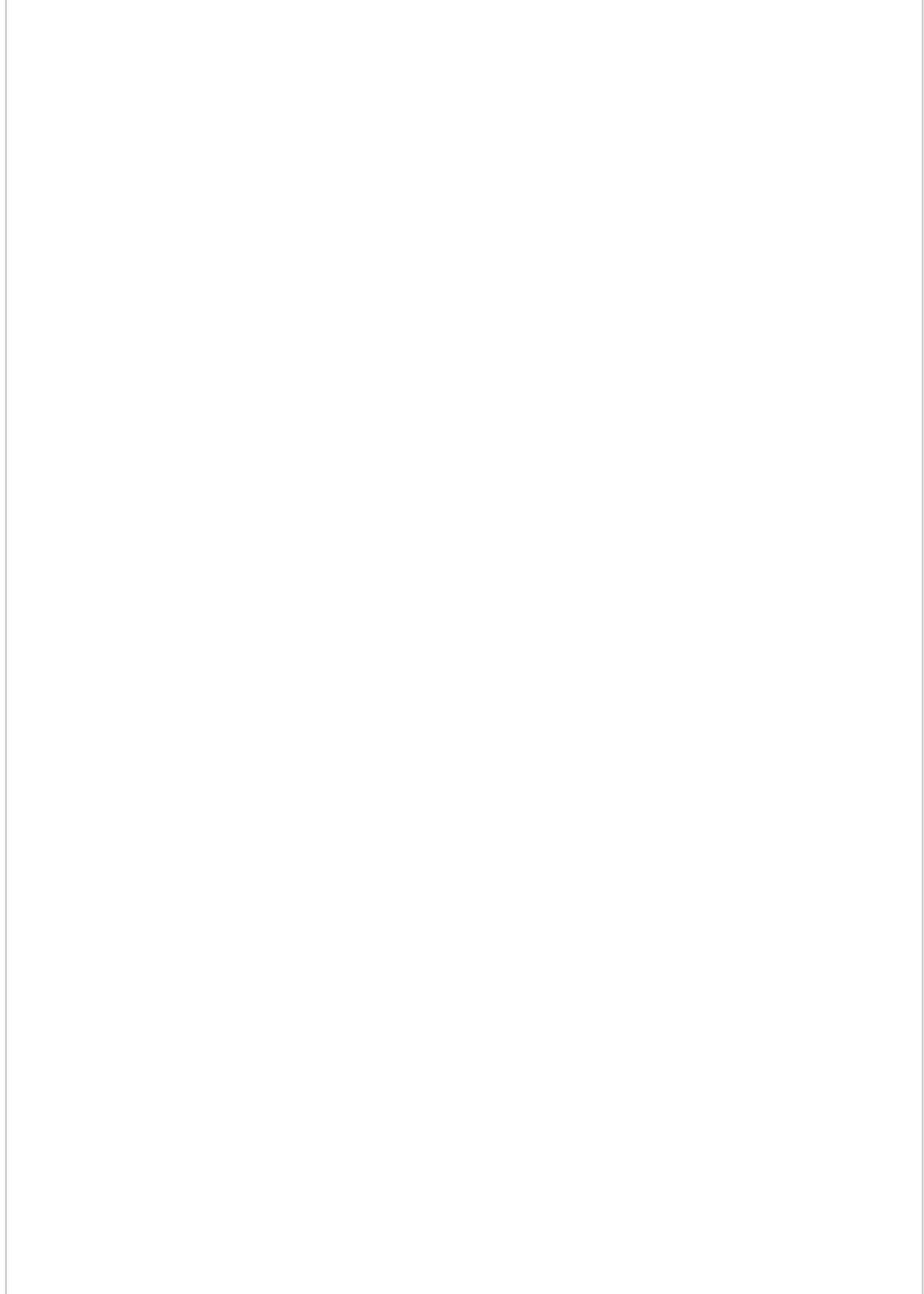


Figure 7-8. Device Farmer user interface. Photo used under Creative Commons.

Device Farmer also helps you to manage your inventory of devices. It shows you which devices are connected, who is using each device, and the hardware spec for your devices, and it assists with physically locating devices in a large lab.

Finally, Device Farmer also has a system for booking and partitioning groups of devices. You can split your device inventory into distinct groups that have owners and associated properties. These groups can then be permanently allocated to projects or organizations or they can be booked for a specific time period.

To set up a device lab you will also need hardware to support the devices. The basic hardware setup includes the following:

- Driver computer - Even though Device Farmer can run on any operating system, it is recommended to run it on a Linux-based host for ease of administration and the best stability. A good option for getting started with this is a compact, but powerful, computer like the Intel NUC.
- USB hub - Both for device connectivity and also to supply stable power, a powered USB hub is recommended. Getting a reliable USB hub is important since this will affect the stability of your lab.
- Wireless router - The devices will get their network connectivity from a wireless router, so this is an important part of the device setup. Having a dedicated network for your devices will increase reliability and reduce contention with other devices on your network.
- Android devices - And the most important part, of course, is having plenty of Android devices to test against. Start with devices that are the most common and popular with your target user base and add additional devices to hit the desired test matrix of Android OS versions, screen sizes, and hardware support as discussed in the previous section.
- Plenty of cables - You will need longer cables than usual to do efficient cable management of devices to the USB hub. It is important to leave enough space between individual devices and hardware components in order to avoid overheating.

With a little bit of work you will be able to create a fully automated device lab similar to **Figure 7-9**, which was the world's first conference device lab featured at the beyond tellerrand conference in Düsseldorf, Germany.

Picture of a volunteer built device lab at a conference

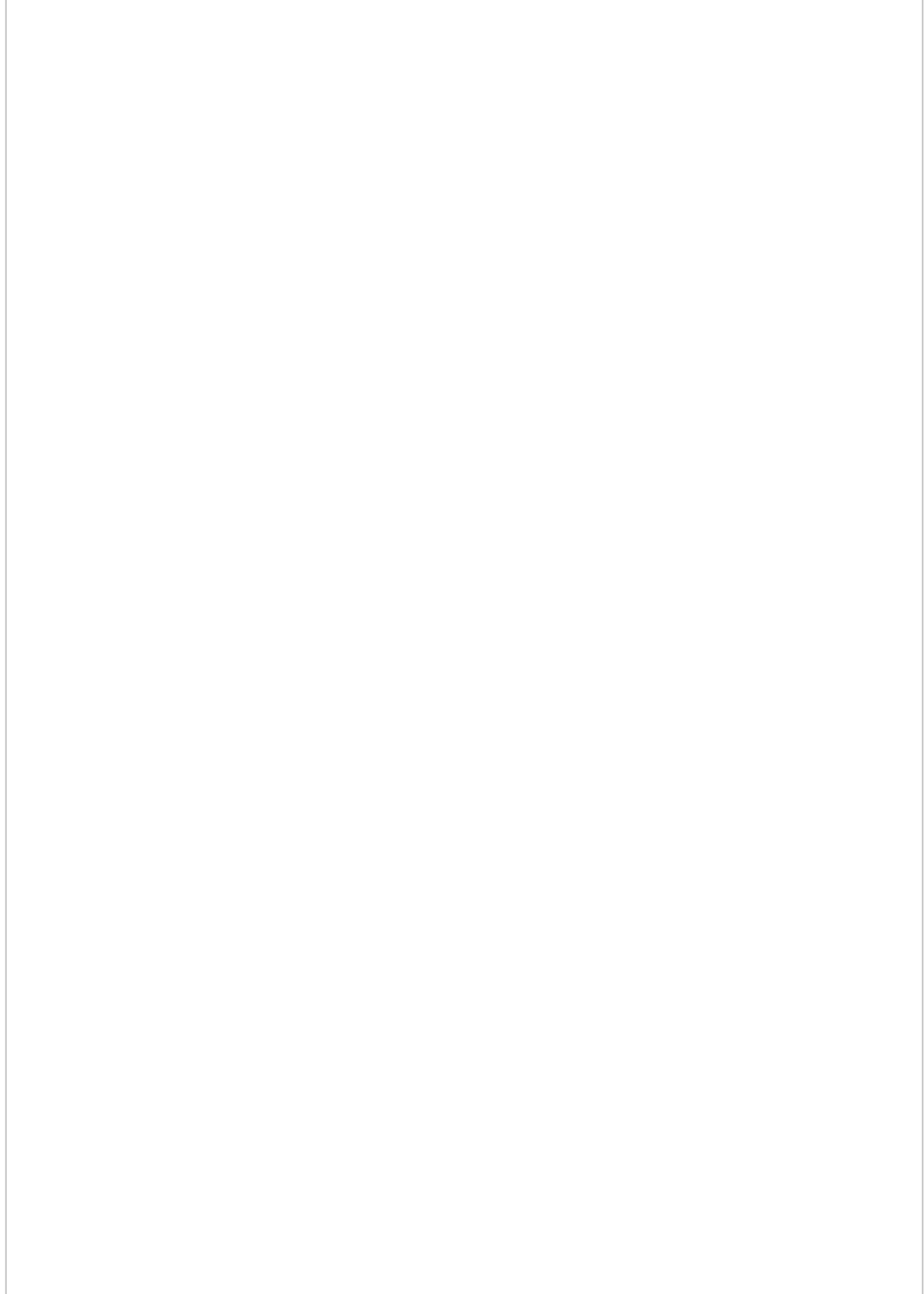


Figure 7-9. *Open device lab* at the beyond tellerrand conference in Düsseldorf, Germany. Photo used under Creative Commons.

Device Farmer is split up into microservices to allow for scalability of the platform to thousands of devices. Out of the box you can easily support 15 devices, after which you will run into port limitations with ADB. This can be scaled out by running multiple instances of the Device Farmer ADB and Provider services up to the limit of the number of USB devices that your machine can support. For Intel architectures this is 96 endpoints (including other peripherals) and for AMD you can get up to 254 USB endpoints. By using multiple Device Farmer servers you can scale into the thousands of devices, which should be enough to support mobile testing and verification of enterprise Android applications.

One example of a large-scale mobile device lab is Facebook's mobile device lab at its Prineville, Oregon data center shown in **Figure 7-10**. They build a customer server rack enclosure for holding mobile devices that is designed to block Wi-Fi signals to prevent interference between devices in their data center. Each enclosure can support 32 devices and is powered by 4 OCP Leopard servers that connect to the devices. This provides a stable and scalable hardware setup that allowed the company to reach to its target device farm size of 2000 devices.

Photo of a data center holding mobile devices in rack enclosures.

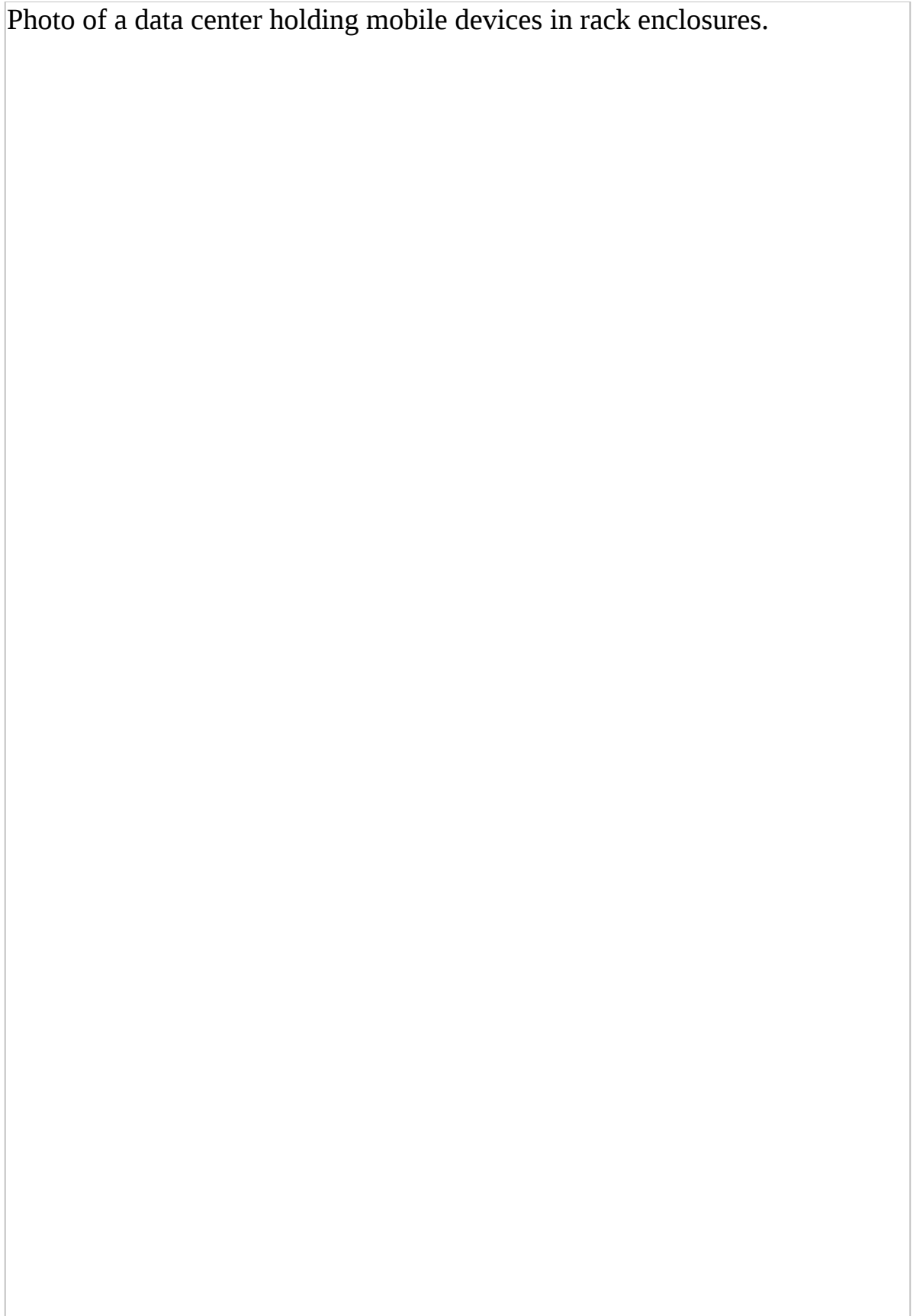


Figure 7-10. The Facebook mobile device lab in their Prineville data center³

There are some challenges of running a large scale device lab:

- *Device maintenance*: Android devices are not meant to be run 24/7 for automated testing. As a result, you are likely to experience higher than normal device failure and have to replace batteries or entire devices every year or two. Spacing out devices and keeping them well cooled will help with this.
- *Wi-Fi interference/connectivity*: Wi-Fi networks, especially consumer targeted Wi-Fi routers, are not highly stable especially with a large number of devices. Reducing the broadcast signal power of the Wi-Fi routers and making sure they are on non-competing network bands can reduce interference.
- *Cable routing*: Running cables between all the devices and the USB hubs or computers can create a tangled mess. Besides being hard to maintain, this can also cause connectivity and charging issues. Make sure to remove all loops in the cables and use shielded cables and ferrite cores as necessary to reduce electromagnetic interference.
- *Device reliability*: Running a device lab on consumer devices comes along with the general risk that consumer devices are not reliable. Limiting automated test runs to a finite duration will help prevent tests from becoming blocked on non-responsive devices. Between tests some housekeeping to remove data and free memory will help with performance and reliability. Finally, both the Android devices and also the servers running them will need to be rebooted periodically.

TIP

It is easy to get started on a small scale with devices you already own and can improve the ability to test across a range of devices and fire off automated test in parallel. At large scale this is an effective solution to solve testing across the fragmented Android ecosystem, but comes with high up-front costs and ongoing support and maintenance.

The next section talks about device labs that you can get started with today on a simple pay as you go basis.

simple pay-as-you-go basis.

Mobile Pipelines in the Cloud

If the prospect of building your own device lab seems daunting, an easy and inexpensive way to get started with testing across a large range of devices is to make use of a device farm running on public cloud infrastructure. Mobile device clouds have the advantage of being easy to get started with and maintenance free for the end user. You simply select the devices you want to run tests on and fire off either manual or automated tests of your application against a pool of devices.

Some of the mobile device clouds also support automated robot tests that will attempt to exercise all the visible UI elements of your application to identify performance or stability issues with your application. Once tests are run you get a full report of any failures, device logs for debugging, and screenshots for tracing issues.

There are a large number of mobile device clouds available with some dating back to the feature phone era. However, the most popular and modern device clouds have ended up aligning with the top three cloud providers, Amazon, Google, and Microsoft. They all have sizeable investments in mobile test infrastructure that you can try for a reasonable price and have a large range of emulated and real devices to test against.

AWS device farm

Amazon offers a mobile device cloud as part of its public cloud services. Using AWS Device Farm you can run automated tests on a variety of different real world devices using your AWS account with.

The steps to create a new AWS Device Farm test are as follows:

1. *Upload your APK file:* To start, upload your compiled APK file or choose from recently updated files.
2. *Configure your test automation:* AWS Device Farm supports a variety of different test frameworks including Appium tests (written in Java, Python, Node.js, or Ruby), Calabash, Espresso, Robotium, or UI Automator. If you don't have automated tests, they provide two

different robot app testers called Fuzz and Explorer.

3. *Select devices to run on:* Pick the devices that you want to run your test on from a user created pool of devices or their default pool of the five most popular devices as shown in **Figure 7-11**.
4. *Setup the device state:* To setup the device before starting the tests, you can specify data or other dependent apps to install, set the radio states (Wi-Fi, Bluetooth, GPS, and NFC), change the GPS coordinates, change the locale, and setup a network profile.
5. *Run your test:* Finally, you can run your test on the selected devices with a specified execution timeout of up to 150 minutes per device. If you tests execute more quickly this can finish earlier, but this also sets a maximum cap on the cost of your test run.

Screenshot of creating a new run on the AWS Device Farm.

Figure 7-11. Configuration for selecting devices to run on in the AWS Device Farm wizard

AWS Device Farm offer a free quota for individual developers to get started with test automation, low per-minute pricing for additional device testing, and monthly plans to do parallel testing on multiple devices at once. All of these plans operate on a shared pool of devices, which at the time of writing included 91 total devices, 54 of which were Android devices, as shown in [Figure 7-12](#). However, most of these devices were highly available, indicating that they had a large number of identical devices to test against. This means that you are less likely to get blocked in a queue or have a device you need to test against become unavailable.

Screenshot of a table of devices.

Figure 7-12. List of available devices in the AWS Device Farm

Finally, AWS Device Farm offers a couple integrations to run automated tests. From within Android Studio you can run tests on the AWS Device Farm using its Gradle plug-in. If you want to launch AWS Device Farm tests from your continuous integration system, Amazon offers a Jenkins plugin that you can use to start device tests right after your local build and test automation completes.

Google Firebase Test Lab

After Google's acquisition of Firebase, it has been continually expanding and improving the offering. Firebase Test Lab is its mobile device testing platform that provides very similar functionality to AWS Device Farm. To get started, Google offers a free quote for developers to run a limited number of tests per day. Beyond that you can upgrade to a pay-as-you-go plan with a flat fee per device hour.

Firebase Test Lab offers several different ways you can fire tests off on the service:

1. **Android Studio** - Firebase Test Lab is integrated in Android Studio and allows you to run tests in their mobile device cloud just as easily as you would on local devices.
2. **Firebase Web UI** - From the Firebase web console you can upload your APK and will start by running your first app in an automated Robo test as shown in **Figure 7-13**. In addition, you can run your own automated tests using Espresso, Robotium, or UI Automator. For game developers there is also an option to run an integrated game loop that simulates user scenarios.
3. **Automated Command Line Scripts** - You can easily integrate Firebase Test Lab into your continuous integration system using its command-line API. This allows you to integrate with Jenkins, CircleCI, JFrog Pipelines, or your favorite CI/CD system.

Screenshot of the Firebase web console testing an app.

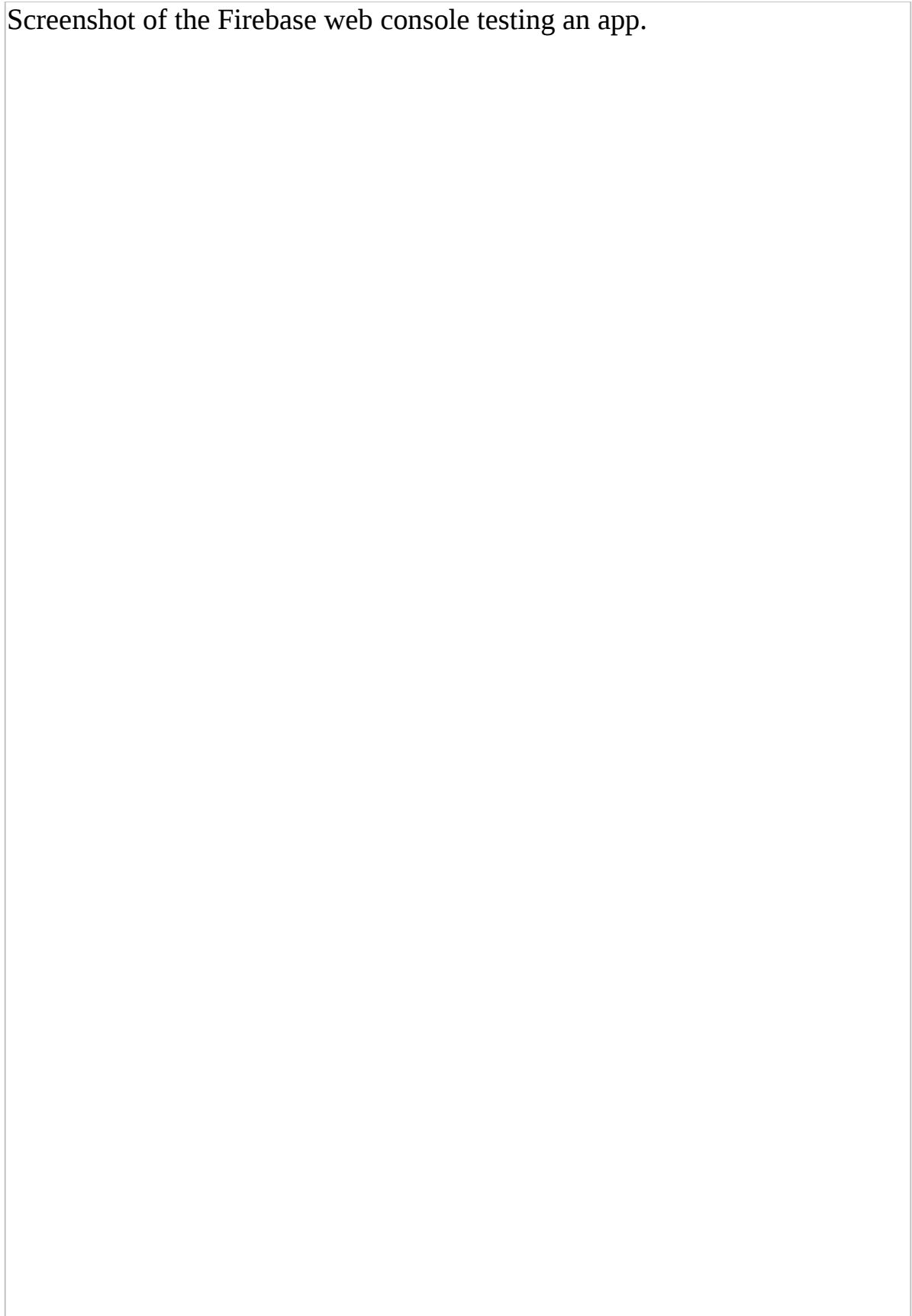


Figure 7-13. Firebase web user interface running an automated Robo test

As of the time of writing, Firebase Test Lab offered a larger collection of Android devices than AWS Test Lab with 109 devices supported, as well as multiple API levels for popular devices. Given the tight integration with Google's Android tooling and also the generous free quota for individuals, this is an easy way to get your development team started building test automation.

Microsoft Visual Studio App Center

Microsoft Visual Studio App Center, formerly Xamarin Test Cloud, offers the most impressive device list of any of the clouds, with 349 different Android device types for you to run tests on, as shown in **Figure 7-14**. However, unlike AWS Device Farm and Firebase Test Lab, there is no free tier for developers to use the service. What Microsoft does offer is a 30 day trial on its service to use a single physical device to run tests, and paid plans where you pay by the number of concurrent devices you want to use, which makes sense for large enterprises.

Screenshot of the device list in VS App Center.

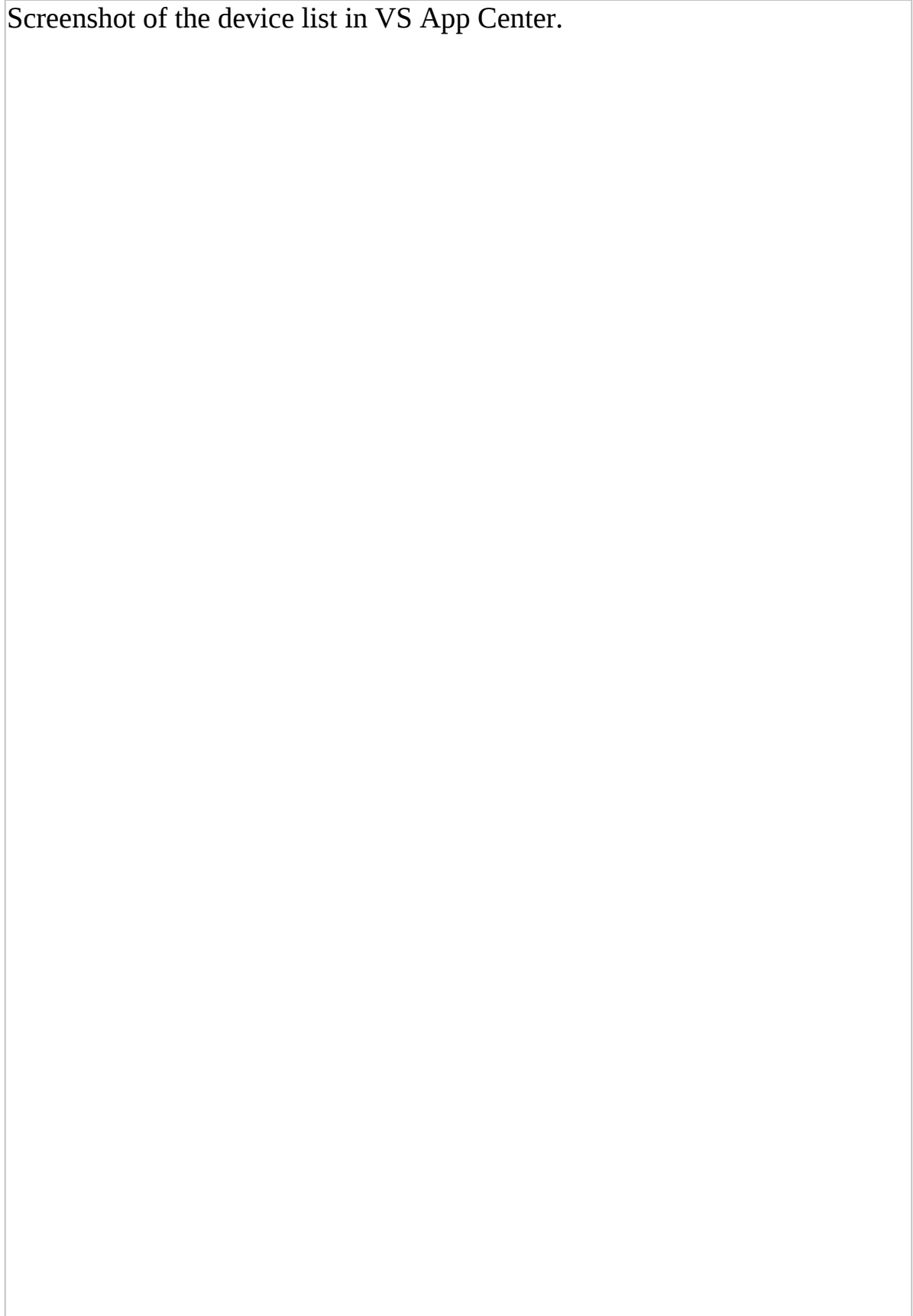


Figure 7-14. Visual Studio App Center device selection screen

Visual Studio App Center also is missing some of the user friendly features like a robot tester and simple test execution via the web console. Instead it focuses on the command-line integration with the App Center Command Line Interface (CLI). From the App Center CLI you can easily fire off automated tests using Appium, Calabash, Espresso, or XamarinUITest. Also, this makes integration with CI/CD tools straightforward.

Overall, Visual Studio App Center wins on device coverage and has a clear focus on Enterprise mobile device testing. However, for independent developers or smaller teams it is less approachable and has higher up-front costs, but will work well as you scale.

Planning a Device Testing Strategy

Now that you've seen the basics of both setting up your own device lab and leveraging cloud infrastructure, you should have a better idea about how these map on to your mobile device testing needs.

These are advantages of going with a cloud service:

1. *Low startup costs:* Cloud plans often offer a limited number of free device tests for developers and utilization based pricing for testing on devices. When starting out with device testing this is the easiest and least costly way to begin exploring manual and automated device testing.
2. *Large selection of devices:* Since cloud testing providers support a large installed base of customers they have a huge inventory of current and legacy phones to test against. This makes it possible to precisely target the device types, profiles, and configurations that your users are most likely to have.
3. *Fast scaleout:* App development is all about viral marketing and scaling quickly. Rather than investing in costly infrastructure upfront, cloud services allow you to scale up the testing as the size and popularity of your application requires a larger device test matrix.

4. *Reduced capital expenditures:* Building a large device lab is a costly upfront capital expenditure. By paying as you go for cloud infrastructure you can delay the costs, maximizing your capital efficiency.
5. *Global access:* With remote and distributed teams becoming the norm, clouds by design allow for easy access from your entire team no matter where they are located.

However, even given all of these benefits, the traditional approach of building a device lab has some unique advantages. Here are some reasons you may want to build your own device lab:

1. *Reduced cost at scale* - The total cost of ownership for a device lab that you run and maintain at scale is much lower than the total monthly costs from a cloud provider over the device's usable lifetime. For a small team this threshold is hard to hit, but if you are a large mobile corporation this can be significant savings.
2. *Fast and predictable cycle time* - With control over the device farm you can guarantee that the tests will run in parallel and complete in a predictable timeframe to enable responsive builds. Cloud providers have limited device availability and queued wait times for popular configurations that can limit your ability to iterate quickly.
3. *No session limits* - Device clouds typically put hardcoded session limits on their service to prevent tests from hanging due to test or device failure. As the complexity of your test suite grows, a 30 minute hard limit can become an obstacle to completing testing of a complex user flow.
4. *Regulatory requirements* - In certain regulated industries such as finance and defense, security requirements can restrict or prohibit the ability to deploy applications and execute tests outside of the corporate firewall. This class of corporations would require an on-premise device lab setup.
5. *IoT device integration* - If your use case requires the integration of mobile devices with IoT devices and sensors, this is not a configuration

cloud providers would provide as a service out of the box. As a result you are probably better off creating a device lab with the IoT and mobile configuration that best matches your real world scenario.

TIP

In some scenarios it also makes sense to do a mix of both cloud testing and local device lab testing. Based on your specific requirements for cycle time, maintenance cost, device scale-out, and regulatory requirements, this can allow you to get the best of both approaches to testing.

Summary

Android is the most popular mobile platform on the planet, because of the huge ecosystem of manufacturers and application developers. However, this is also the challenge with Android development, because it has produced an incredibly fragmented device market with thousands of manufacturers producing tens-of-thousands of devices. Given the scale of fragmentation and device inconsistency in the mobile space, having a fully automated DevOps pipeline for mobile development is a necessity for success.

The equivalent to DevOps for web application development would be if instead of three major browsers there were thousands of unique browser types. You would be forced to automate to obtain any level of quality assurance, which is exactly why there is so much focus in the mobile space on UI test automation running on real devices.

Using the tools and techniques you learned in this chapter, paired with the overall DevOps knowledge on source control, build promotion, and security, you should be ahead of your mobile DevOps peers to face the challenge of continuous deployments to millions of devices globally.

1 <https://www.bidouille.org/misc/androidcharts>

2 <https://blog.appliedmaterials.com/think-your-smartphone-resolution-good-think-again>

3 Antoine Reversat (<https://engineering.fb.com/2016/07/13/android/the-mobile-device-lab-at-the-prineville-data-center/>), “The mobile device lab at the Prineville data center”

Chapter 8. Continuous Deployment Patterns and Antipatterns

Stephen Chin

Baruch Sadogursky

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors’ raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 10th chapter of the final book. If there is a GitHub repo associated with the book, it will be made active after final publication.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at ccollins@oreilly.com.

“Learn from the mistakes of others. You can’t live long enough to make them all yourself.”

—Eleanor Roosevelt

In this chapter we will give you the patterns for continuous deployment that you need to be successful with implementing DevOps best practices in your organization. It is important to understand the rationale for continuous updates to convince others in your organization about the change needed to improve your deployment process.

However, we will also give you plenty of antipatterns from companies that have failed to adopt continuous update best practices. It is always better to learn from the failures of others, and there are plenty of recent examples in the high technology industry of what not to do and the consequences of ignoring best practices.

practices.

By the time you complete this chapter you will be armed with knowledge of seven best practices of continuous updates that you can start using today in order to join the top 20% DevOps “Elite Performers” of the software industry¹.

Why Everyone Needs Continuous Updates

Continuous updates are no longer an optional part of a software development, but are now a required part of any major project. Planning for continuous delivery of updates is just as important as the functional requirements of the project and requires a high level of automation to execute on reliably.

It was not always this way. Historically, software was delivered on a much lower cadence and only received critical updates. Also, installation of updates was often a manual and error prone process that involved tweaking of scripts, data migration, and significant downtime.

This has all changed in the past decade. Now end users have an expectation that new features get added constantly, which is driven by their experience with consumer devices and continuously updated applications. Also, the business risk associated with deferring critical updates is significant as security researchers constantly uncover new exploits that can be used to compromise your system unless it is patched. Finally, this has become a business expectation in the cloud age where the entire infrastructure stack is constantly being updated to improve security, often with the requirement that you also update your application.

Not all software projects have been as quick to adopt continuous update strategies, especially in industries that are used to longer technology adoption cycles. However, the widespread use of common hardware architectures and open source technologies means that these projects are at an equal risk of exposure from critical vulnerabilities. When exposed, this can lead to catastrophic failures that are difficult or impossible to recover from.

In the next few sections we will dig into the motivation for continuous updates in more detail. If you do not already have a continuous update strategy, this will help you to convince others in your organization to adopt this. Or if you have already embraced continuous updates you will be armed with knowledge on how to reap the business benefits of having superior infrastructure and DevOps

processes to your competitors.

User Expectations on Continuous Updates

The expectation of end users on release cadence of new features has dramatically shifted in the last decade. This is driven by a change in how features and updates are delivered on consumer devices, but translates to similar expectations on other software platforms, even in the enterprise. Forcing users to wait for a long release cycle or to perform a costly migration to take advantage of new users will result in dissatisfied users and put you at a competitive disadvantage.

Continuous Updates in the Consumer Industry

This change in user expectations can be seen in several consumer industries including cell phones. When mobile communication first started to gain popularity, Nokia was one of the dominant hardware manufacturer of 2G cell phones. While primitive by today's standards, they had excellent hardware design with good voice quality, tactile buttons, and rugged design as demonstrated by the very popular Nokia 6110 shown in **Figure 8-1**.

An image of a Nokia 6110 cell phone

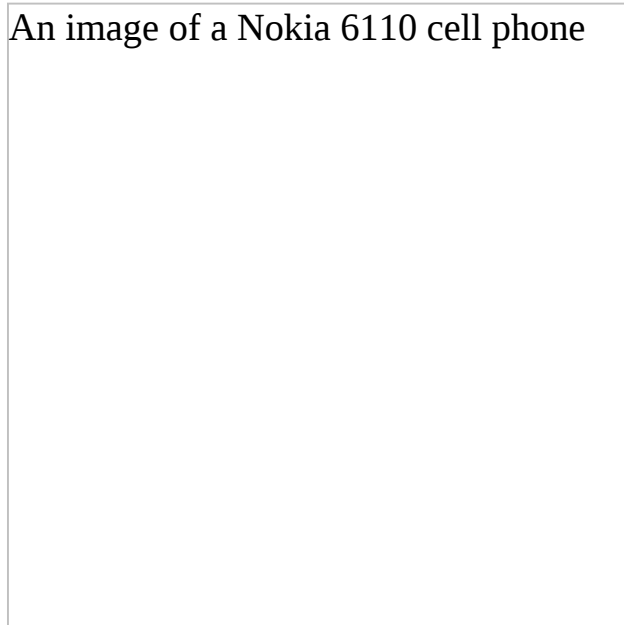


Figure 8-1. title The Nokia 6110 phone²

Small form factor mobile devices such as the Nokia 6110 accelerated adoption of cellular technology, but the software on these devices and users' ability to update them was extremely poor. This was a common problem with early

consumer device companies in that they considered themselves a hardware company first and were slow to adopt modern practices in software development.

Like many emerging technologies, the software shipped with Nokia phones was bare bones and buggy, requiring patches and updates to be usable. While they offered a data cable, this was limited to basic operations like transferring contacts from the device to a computer, but not maintenance features such as performing firmware updates. To get a feature update on your phone that contained some important patches and mission critical features (like the Snake game), you would need to take your phone in to a service center to have them update your device as shown in **Figure 8-2**.

Figure 8-2. Progression of software updates on mobile devices

It wasn't until the iPhone came out in 2007 that the phone industry took a software first approach to mobile phone design. With the ability to update the firmware and entire operating system from an attached computer and later over-the-air updates, Apple could rapidly deploy new features to existing devices.

In 2008 Apple announced the app store, which created a vibrant app ecosystem and laid the foundation for modern store features like security sandboxing and automatic application updates, which we will come back to later in this chapter with a longer case study. With the release of iOS 5 in 2011, Apple embraced over-the-air updates where you no longer even needed a computer to install the latest version of the operating system.

Now the process of updating software on your phone is seamless and automated to the point where most consumers have no idea what version of the operating system or individual applications they are running. As an industry we have trained the general public that continuous updates are not only expected, but required for functionality, productivity, and security.

This model of continuous updates has become the norm for consumer devices of all types including smart TVs, home assistants, and even newer self-updating routers. While the car industry has been slow to adopt a continuous update strategy, Tesla is pushing the industry with bi-weekly updates to your vehicle right on your home network. No longer do you need to drive to a vehicle service center for a recall or critical software update.

Security Vulnerabilities Are the New Oil Spills

Oil spills have had a tremendously detrimental effect on the environment over the past 50 years and continue to be an ongoing crisis. When running smoothly, oil drilling rigs are immensely profitable, but when there are accidents or natural disasters, particularly at sea where environmental damage is amplified, the cost can be enormous. For large companies like BP, who can afford to pay or set aside tens of billions of dollars for fines, legal settlements, and cleanups, oil spills are just a cost of doing business. However, for drilling operations run by smaller companies a single oil spill can spell financial disaster and put companies out of business with no means to address the aftermath.

Illustration of the Tayler Energy oil platform

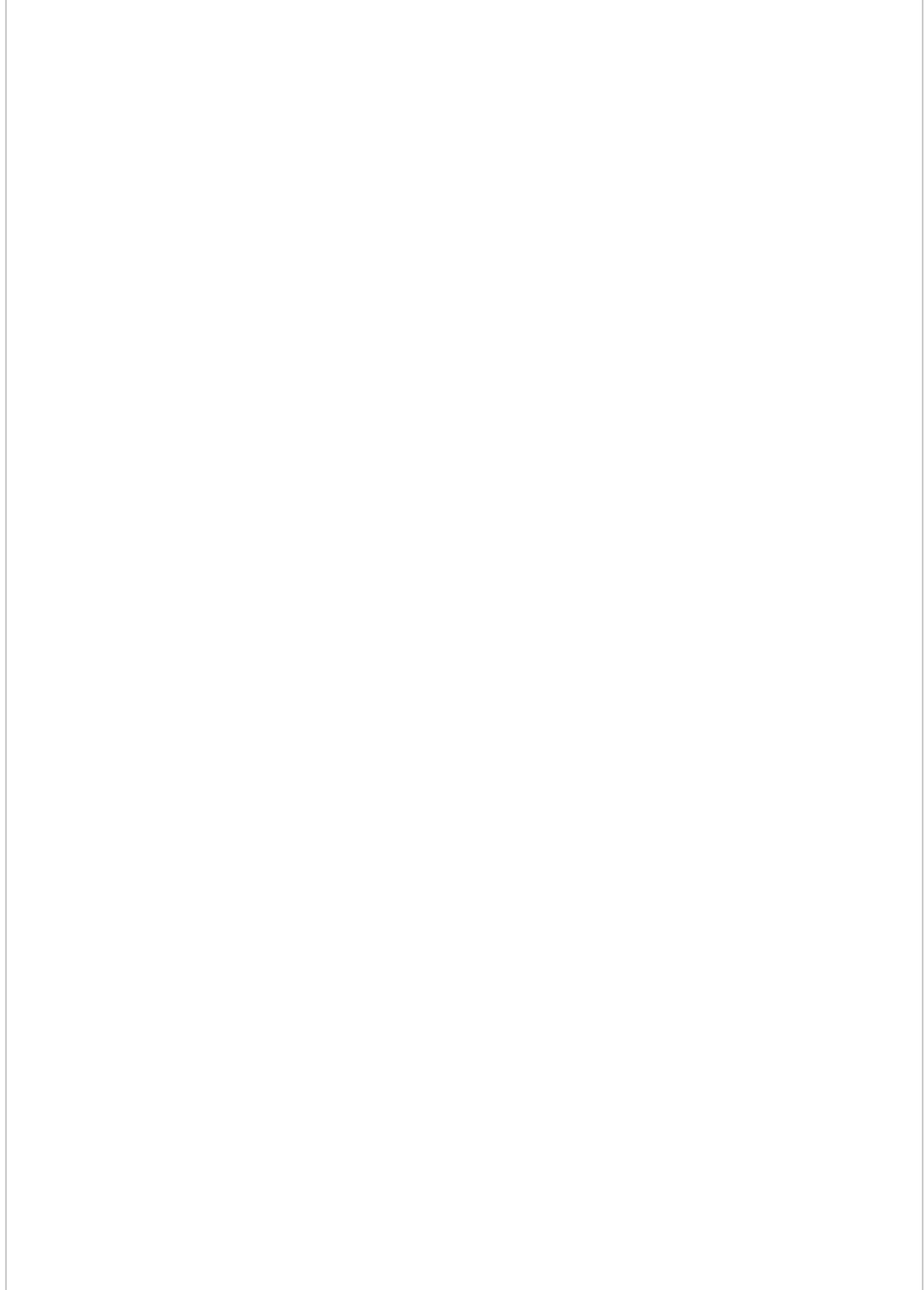


Figure 8-3. Tayler Energy offshore oil platform prior to collapse³

This was the case for Tayler Energy who lost the oil platform shown in **Figure 8-3** off the coast of Louisiana to Hurricane Ivan in 2004 and has been leaking 300 to 700 barrels per day⁴. This disaster continues to haunt Tayler Energy who is both the recipient and instigator of multiple lawsuits surrounding the oil spill and ongoing containment efforts. Taylor Energy has already spent \$435m to reduce the oil leakage in what has become the longest oil spill in U.S. history and has the potential to keep leaking for the next century⁵.

This is analogous to the risk that software vulnerabilities pose to the technology industry. Software systems have become increasingly complex, which means there are more dependencies on open source software and third-party libraries, making it is virtually impossible to audit and guarantee that a system is free of security vulnerabilities.

According to the 2020 Open Source Security and Risk Analysis report by Synopsis⁶, open source software is used in 99% of enterprise projects and 75% of those projects contained at least one public vulnerability with an average of 82 vulnerabilities found per codebase.

So how bad are these vulnerabilities that plague commercial codebases? The top 10 vulnerabilities allow for for an attacker to obtain sensitive information like authentication tokens and user session cookies, execute arbitrary code in the client browser, and trigger denial-of-service conditions.

Organizations' reactions to security vulnerabilities can be organized into three discrete steps that have to occur sequentially in order to respond:

- Identify
 - First the organization must realize that a security issue exists and is currently or can potentially be exploited by an attacker.
- Fix
 - Once a security issue is identified, the development team must come up with software fix to patch the issue.
- Deploy

- The final step is to deploy the software fix that addresses the security issue, often to a large set of end users or target devices that are affected by the vulnerability.

Going back to the Tayler Energy oil spill, you can see how difficult these steps are in the physical world:

- Identify - 6 Years

- In 2004 Hurricane Ivan destroyed the offshore platform and the resulting landslide buried 28 oil wells 475 feet below the sea floor. This made identification of the full scope of the disaster challenging, and it wasn't until six years later in 2010 that researchers observed a persistent oil slick at the Taylor site and brought it to public attention. However, without the technology available to cap off the buried oil wells, containment efforts were stalled.

- Fix - 8 Years

- It wasn't until two separate studies in 2017 identified that oil and gas were discharging from multiple discrete locations rather than a wide area that it became possible to devise a fix. A follow-up study in 2018 was able to quantify the flow rate at 1,000 gallons of oil per day (compared to the 3-5 gallons originally assessed). The Coast Guard took over the cleanup efforts and put out a bid for a containment system that was won by the Couvillon Group who proposed an underwater shield to collect the oil.

- Deploy - 5 Months

- In April 2019 the Couvillon Group deployed a shallow 200-ton steel box containment system above the discharge area as shown in **Figure 8-4**. While not a permanent fix, this containment system has been collecting about 1,000 gallons of resellable oil per day and reduced the visible pollutants on the ocean surface.

Illustration of the Couvillon Group oil containment system

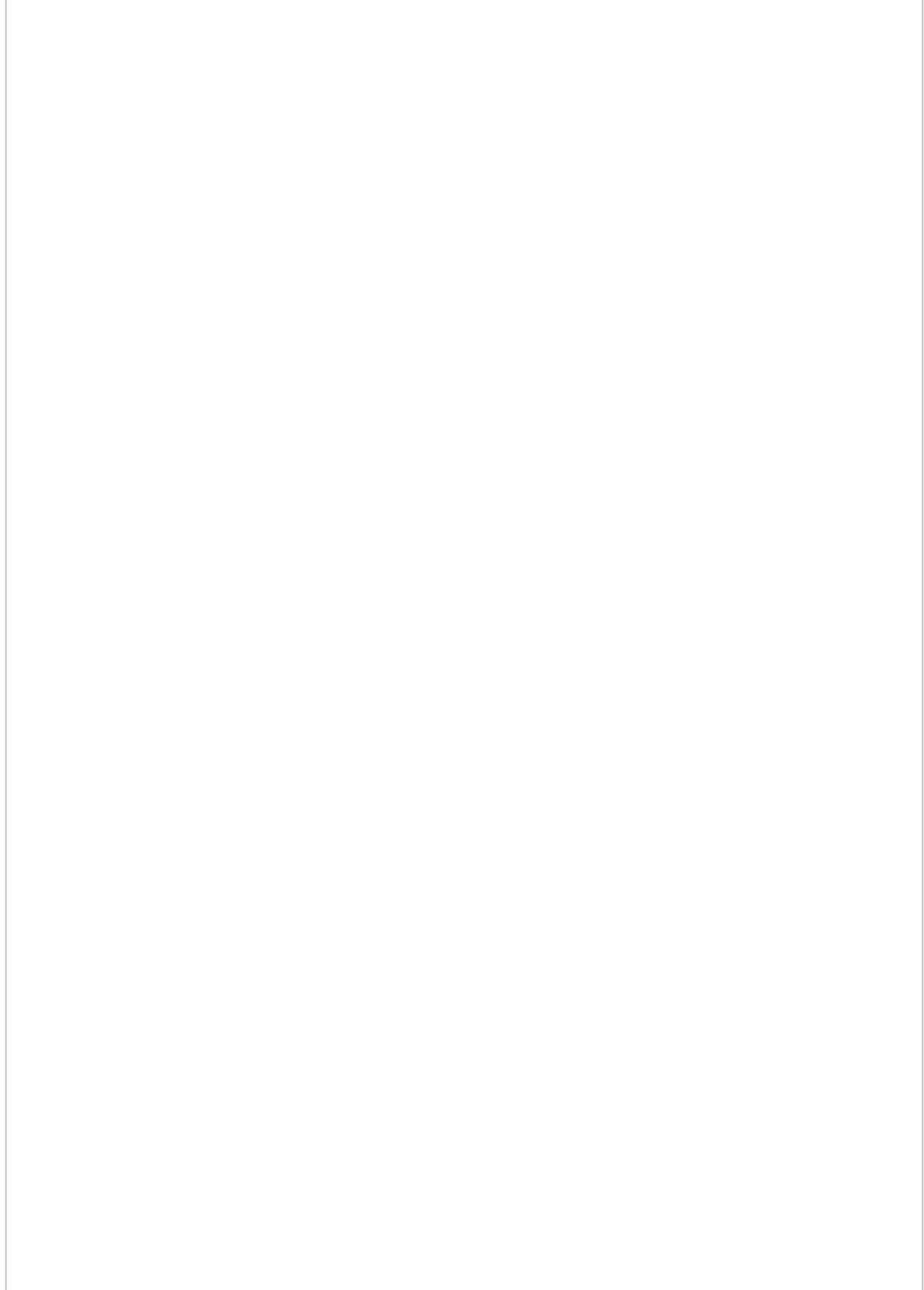


Figure 8-4. Oil platform after collapse with containment system⁷

Compared to a physical disaster like this, you would think that security vulnerabilities would be relatively easy to identify, fix, and deploy. However, as we will see in the following case studies, software vulnerabilities can be just as damaging and economically costly, and are by far much more common.

UK Hospital Ransomware

In 2017 a worldwide cyberattack was launched that encrypts the hacked computers and requires a bitcoin “ransom” payment to recover the data. This attack exploited the EternalBlue exploit on the Windows Server Message Block (SMB) service that was previously discovered by the NSA and leaked a year prior to the attack.

Screenshot from a wannacry infected computer

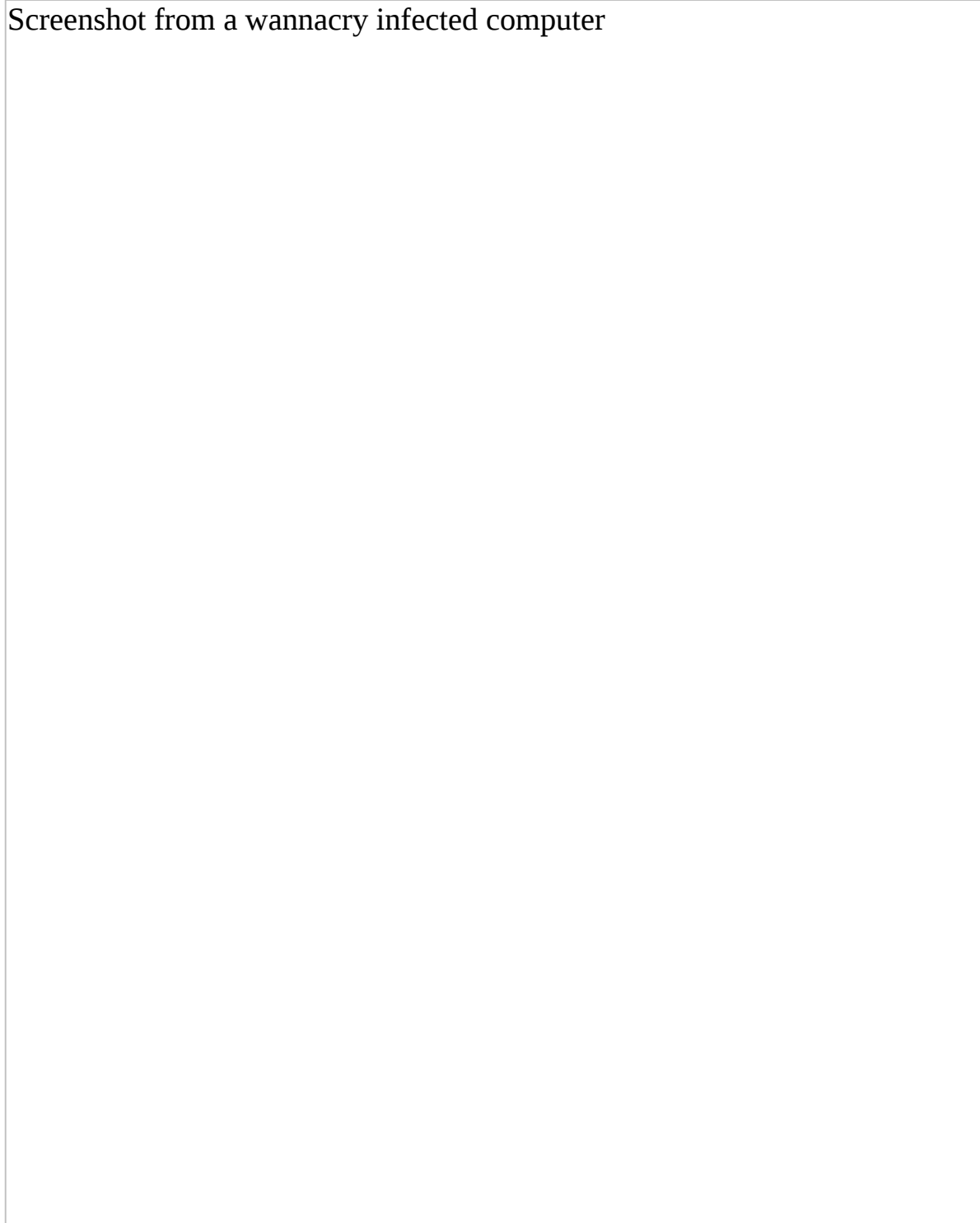


Figure 8-5. Wannacry ransomware screen from an infected computer⁸

Upon infection, the virus attempts to replicate itself on the network and encrypts critical files preventing their access, presenting a ransom screen as shown in **Figure 8-5**. Microsoft had released patches for older version of Windows that

were affected by this exploit, but many systems were not updated due to poor maintenance or a requirement for 24/7 operation.

One organization that was critically impacted by this ransomware attack was the United Kingdom National Health Service (NHS) hospital system. Up to 70,000 devices on their network including computers, MRI scanners as shown in **Figure 8-6**, blood-storage refrigerators, and other critical systems were affected by the virus.⁹ This also involved diversion of emergency ambulance services to hospitals and at least 139 patients who had an urgent referral for cancer that got cancelled.

Photo depicting an MRI scanner

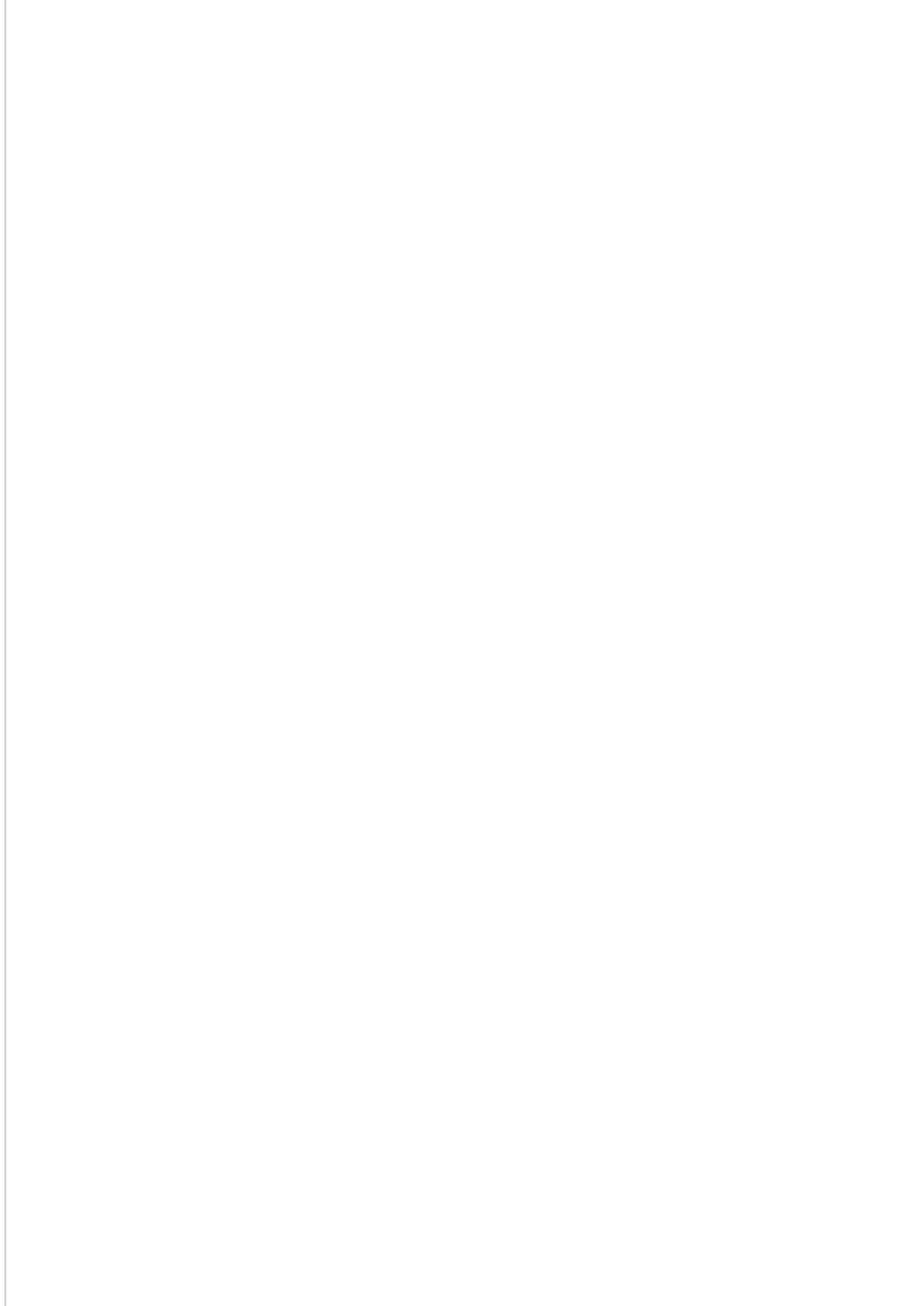


Figure 8-6. An MRI scanner similar to the ones compromised by the UK Hospital ransomware attack¹⁰

The Wannacry ransomware resulted in an estimated 19,000 cancelled appointments and cost approximately £19 million in lost output and £73 million in IT costs to restore systems and data in the weeks after the attack.¹¹ All of the affected systems were running an unpatched or unsupported version of Windows that was susceptible to the ransomware. The majority were on Windows 7, but many were also on Windows XP, which had been unsupported since 2014; a full 3 years prior to the attack.

If we frame this with our vulnerability mitigation steps, we get the following timelines and impacts:

- Identify - 1 Year
 - Both the existence of the vulnerability and an available patch were available for a year preceding the incident. It wasn't until the attack was launched on the world and affected the NHS that this vulnerability was discovered.
- Fix - Existing
 - Since the fix is simply to upgrade or patch systems with an existing fix, this was immediately available by the time the vulnerability was identified.
- Deploy - Multiple Years
 - While critical systems were brought back online quickly, there were enough affected systems that it took several years for the NHS to fully upgrade and patch affected systems with multiple failed security audits.

In this case it was an operating system level security breach. Assuming you are following industry best practices and keeping your operating system under maintenance and continually patched you might believe you are safe, but what about application level security vulnerabilities? This is by far the more common type of security vulnerability and is equally easy to exploit by an attacker as happened to Equifax.

Equifax Security Breach

The Equifax security breach is a textbook example of an application level security vulnerability causing massive financial damage to a high tech company. During the time period from March through July of 2017 hackers had unrestricted access to Equifax's internal systems and were able to extract personal credit information for half of the total U.S. population or 143 million consumers.

This had the potential for massive identity theft, but none of the stolen Equifax personal data appeared on the dark web, which is the most direct monetization strategy. It is instead believed the data was instead used for international espionage by the Chinese government. In February of 2020, four Chinese backed military hackers were indicted in connection with the Equifax security breach as shown in **Figure 8-7**.

IBM wanted poster of Chinese backed military hackers

Figure 8-7. Chinese hackers suspected of infiltrating Equifax's systems¹²

For a credit agency to have a security vulnerability of this magnitude, the damage to their brand and reputation is incalculable. However, it is known that Equifax spent 1.4 billion on cleanup costs¹³ and an additional 1.38 billion to resolve consumer claims. Also, all of the upper executives at Equifax were quickly replaced after the incident.

There were multiple, compounded security vulnerabilities that lead to this breach. The first and most egregious was an unpatched security vulnerability in Apache Struts that allowed hackers to gain access to Equifax's dispute portal. From here they moved to multiple other internal servers to access databases containing information on hundreds of millions of people.

The second major security vulnerability was an expired public-key certificate that impeded their internal system that inspects encrypted traffic exiting their network. The expired certificate wasn't discovered and renewed until July 29th, at which time they became immediately aware of the obfuscated payloads being used by the attacker to extricate sensitive data.

- Identify - 5 Months
 - The initial security breach occurred on March 10th and while the attackers did not actively start exploiting this security breach until May 13th, they had access to the system for almost 5 months until Equifax became aware of the data exfiltration. It wasn't until July 29th when Equifax fixed their traffic monitoring system that they became aware of the breach.
- Fix - Existing
 - The Apache Struts security vulnerability (CVE-2017-5638¹⁴) was published on March 10th 2017 and fixed by Apache Struts 2.3.32 that was released 4 days prior to the CVE disclosure on March 6th.
- Deploy - 1 Day
 - The vulnerability was patched on July 30th, 1 day after they became aware of the breach.

The Equifax breach is particularly scary since it started with a vulnerability in a widely used Java library that affects a large number of systems across the web. Even a year after the security vulnerability was identified, researchers at the SANS Internet Storm Center found evidence¹⁵ of exploitation attempts looking for unpatched servers or new deployments that had not been secured.

Widespread Chipset Vulnerabilities

Even if you are keeping up on security vulnerabilities in the application and operating system level, there is another class of vulnerabilities that can affect you at the chipset and hardware level. The most widespread recent example of this are the Meltdown and Spectre exploits discovered by Google security researchers¹⁶.

These flaws are so fundamental to the hardware platforms that we use to run everything from cloud workloads to mobile devices that security researchers called them catastrophic. Both exploits take advantage of the same underlying vulnerabilities in how speculative execution and caching interact to get access to data that should be protected.

In the case of Meltdown, a malicious program can get access to data across the machine that it should not have access to, including processes with administrative privileges. This is an easier attack to exploit since it requires no knowledge of the programs you are trying to attack, but also it is easier to patch at the operating system level.

Intel i9-9900K processor

Figure 8-8. Intel i9-9900K Processor with hardware mitigation for Meltdown

Upon the announcement of the Meltdown vulnerability the latest versions of Linux, Windows, and Mac OS X all had security patches to prevent Meltdown from being exploited with some performance loss. In October 2018 Intel announced hardware fixes for their newer chips including Coffee Lake Refresh shown in **Figure 8-8**, Cascade Lake, and Whiskey Lake that address various variants of Meltdown¹⁷.

In contrast, exploiting the Spectre vulnerability requires some specific information about the process being attacked making it a more difficult vulnerability to leverage. However, it is also much trickier to patch, which means that new exploits based on this vulnerability continue to be identified. Also it is more dangerous in cloud computing applications that use VMs since it can be used to induce a hypervisor to provide privileged data to a guest operating system running on it.

The result is that Meltdown and particularly Spectre have opened up a new class of security vulnerabilities that break the principles of software security. It was assumed that if you built a system with the proper security protections in place and could fully verify the correctness of the source code and dependent libraries then it should be secure. These exploits break this assumption by exposing side-channel attacks hiding in the CPU and underlying hardware that require further analysis and software and/or hardware fixes to mitigate.

So getting back to our analysis for the general class of chipset side-channel attacks:

- Identify - As fast as possible
 - While there are generalize fixes for Meltdown and Spectre exploits can occur at any time based on the architecture of your application.
- Fix - As fast as possible
 - A software fix for Spectre often involves specially crafted code to avoid either accessing or leaking information in misspeculation.

- Deploy - As fast as possible
 - Getting the fix into production quickly is the only way to mitigate damage.

Of these three variables, the one that can most easily be shortened is the deployment time. If you don't already have a strategy for continuous updates, this will hopefully give you the impetus to start planning for faster and more frequent deployments.

Getting Users to Update

So we have now hopefully convinced you that continuous updates are a good thing, both from a feature/competitive standpoint as well as for security vulnerability mitigation. However, even if you deliver frequent updates will end users accept and install these updates?

The diagram in **Figure 8-9** models the user flow for deciding whether to accept or reject an update.

Flow diagram showing user update flow

Figure 8-9. User Model for Update Acceptance

The first question for a user is whether they really want the update based on features and/or security fixes. Sometimes this is not a binary decision, because there is a choice to stay on a maintenance line with patches for security, but delay major upgrades that provide larger features but are more risky. This is the model that Canonical uses for Ubuntu releases where long term support (LTS) releases come out once every two years with public support for five years. However, for early adopters there are interim releases every six months that don't receive the same extended support.

The second question is how risky is the update? For security patches or minor upgrades the answer is usually that it is low risk and it is safe to put in production with minimal testing. Typically these changes are small, specifically designed to not touch any external or even internal APIs, and tested to make sure they address the security issue and don't produce undesirable side effects before release.

The other scenario where it may be safe to upgrade is where the party releasing the upgrade verifies that it is a safe upgrade as shown in the third decision box of **Figure 8-9**. This is the model for operating system upgrades, such as iOS, where there are significant changes that cannot be individually verified to be non-breaking. The OS vendor has to spend a significant amount of time testing different hardware combinations, working with application vendors to fix compatibility issues or help them upgrade their apps, and performing user trials to see what issues happen during upgrade.

Finally, if it is both risky and the party producing the release cannot verify the safety, it is up to the recipient of the upgrade to do verification testing. Unless it can be fully automated, this is almost always a difficult and costly process to undertake. If the upgrade cannot be proven to be safe and bug free then the release may get delayed or simply skipped over in the hopes that a later release will be more stable.

Let's look at some real world use cases and see their continuous update strategy.

Case Study: Java Six Month Release Cadence

Java has historically had a very long release cycle between major versions

averaging between 1 and 3 years. However, the release frequency was erratic and often delayed, such as Java 7, which took almost 5 years to be released¹⁸. The release cadence has continued to decline as the platform has grown due to several factors such as security issues and the difficulty of running and automating acceptance tests.

Starting with Java 9 in September of 2017, Oracle made the dramatic move to a six month feature release cycle. These releases can contain new feature and also remove deprecated features, but the general pace of innovation was intended to stay constant. This means that each subsequent release should contain fewer features and less risk, making it easier to adopt. The actual adoption numbers of each JDK release are shown in **Figure 8-10**.

Graph showing percentage adoption of each Java release

Figure 8-10. Developer adoption of recent Java releases¹⁹

Given that 67% of Java developers never made it past Java 8, which came out in 2014, there is clearly a problem in the new release model! However, hidden under the data are a few different issues.

The first is that the Java ecosystem can't handle six month releases. As we learned in the package management chapter, virtually all Java projects are dependent on a large ecosystem of libraries and dependencies. In order to upgrade to a new Java release all of those dependencies need to be updated and tested against the new Java release. For large open source libraries and complex application servers this is almost impossible to accomplish in a six month timeframe.

To compound this, the OpenJDK support model only provides public support for Java releases for six months until the next feature release comes out. This means that even if you could upgrade every six months, you would be left without critical support and security patches as detailed in Stephen Coulbourne's blog²⁰.

The only exception to this is Long Term Support releases that start with Java 11 and come every 3 years thereafter. These releases will get security patches and support from commercial JDK vendors such as Oracle, Redhat, Azul, BellSoft, SAP, and others. There are also some free distributions like AdoptOpenJDK and Amazon Corretto that promise to provide Java releases and security patches at no cost. This is why Java 11 is the most popular release after Java 8 and none of the other six month releases have gained any traction.

However, in comparison to Java 8, Java 11 has not gained significant traction. The number of developers using Java 11, roughly two years after its release in September 2018, is 25%. In contrast, exactly two years after the release of Java 8 the adoption was 64% as shown in **Figure 8-11**. This comparison is also biased in favor of Java 11, because anyone who adopted Java 9 or 10 would likely have upgraded to Java 11, providing 3 full years of adoption growth.

Graph showing growth of Java 8 adoption

Figure 8-11. Developer adoption of Java 8 two years after release²¹

This brings us to the second reason for the poor adoption of Java 9 and beyond, which is a poor value/cost tradeoff. The main feature of Java 9 was the introduction of a new module system. The idea of a modular Java Platform was first suggested by Mark Reinhold back in 2008²² and took 9 years to complete in the release of Java 9.

Due to the complexity and disruptiveness of this change it was delayed several times missing both Java 7 and Java 8 as initial targets. Also it was highly controversial on release because it was initially incompatible with OSGI, a competing module system released by the Eclipse foundation targeted at enterprise applications.

But perhaps the bigger issue with modularity is that no one really was asking for it. There are many benefits to modularity including better library encapsulation, easier dependency management, and smaller packaged applications. However, to fully realize these benefits you need to spend a lot of work rewriting your application to be fully modularized which is a lot of work. Secondly, you need all of your dependencies to be packaged as modules, which has taken a while for open source projects to embrace. Finally, the practical benefits for most enterprise applications are small so even after upgrading to a module enabled release it is very common practice to disable modularity and go back to the classpath model of Java 8 and prior.

Flow diagram showing Java developer update flow

Figure 8-12. User Model for Java Release Acceptance

Getting back to the update diagram, **Figure 8-12** shows the simplified developer thought process on upgrading to Java 9 and beyond. Clearly it comes down to a value comparison between how much you want modularity or other newly introduced features versus how costly it is to upgrade. And the upgrade cost is highly dependent on how difficult it is to test your application after upgrading, which takes us to our first continuous update best practice:

CONTINUOUS UPDATE BEST PRACTICES

- Automated Testing

Automate, automate, automate. The more automated testing you have, and the quicker your tests run, the faster you can adopt new features safely and release into production.

Case Study: iOS App Store

We have had a very different update model for content since 1990 with the creation of the first web browser called WorldWideWeb by Tim Berners-Lee²³. Using a client-server model, content could be retrieved dynamically and updated continuously. As Javascript and CSS technologies matured this turned into a viable app delivery platform for continuously updated applications.

In contrast, while desktop client applications were comparatively complex and rich in their user interface, updates were infrequent and manual. This created a situation up to the mid 2000s where you had to choose between either rich client applications that were difficult to update in the field or simpler web applications that could be continuously updated to add new features or patch security vulnerabilities. If you are a continuous update fan (which you should be by now), you know which one wins.

Picture of the logo used for apps available in the app store

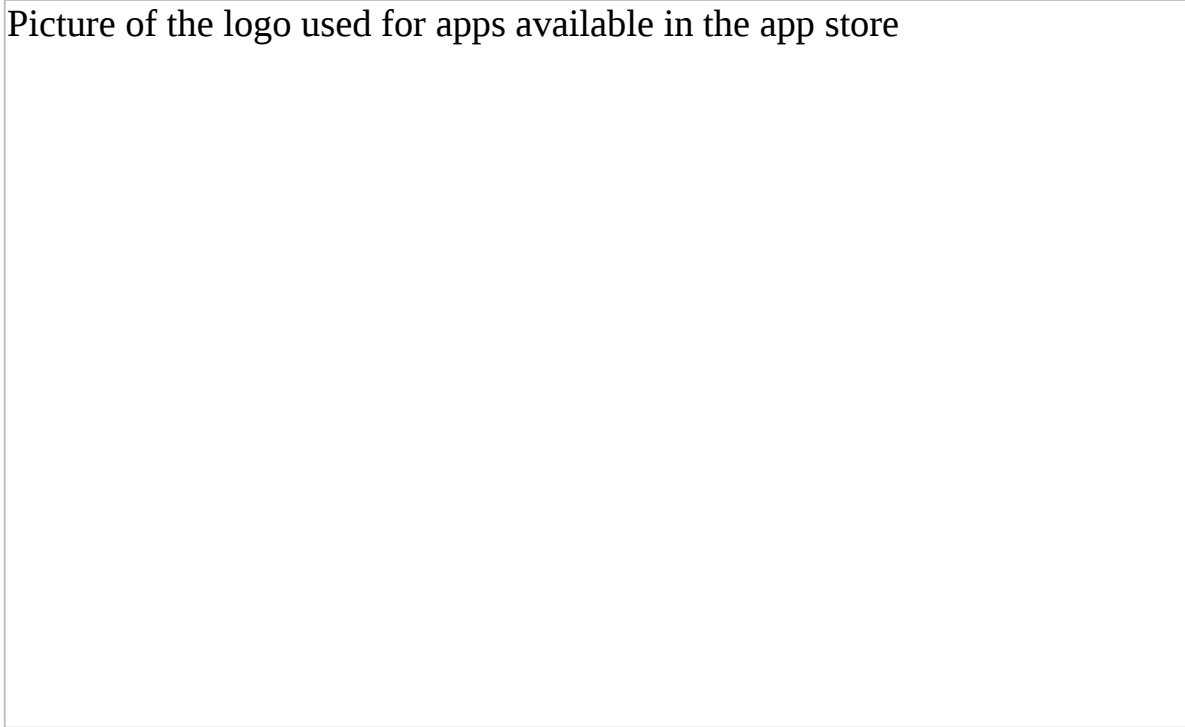


Figure 8-13. Apple App Store logo²⁴

However, Apple changed all of this with the App Store on the iPhone in 2008, which was a game changing method for deploying rich client applications to phones and other devices with the logo shown in **Figure 8-13**:

- Update in one click
 - Updating a desktop application requires quitting the running version, some sort of guided wizard to go through a seemingly dizzying array of choices for the common case (e.g. desktop shortcut, start menu, optional packages), and often rebooting your computer after install. Apple simplified this to a single button update, and in the case of many updates a bulk option to update them all at once. The app update downloads, your mobile app quits, and the app is installed all in the background with no user interruptino.
- There is only one version: latest
 - Do you know what version of Microsoft Office you are running? Up until 2011 when Office 365 was released you had

to, and had likely not upgraded in the past 3 to 5 years (or more). Apple changed all of this by only providing the latest version in the app store so the choice of what version to upgrade to is entirely removed. Also, you are not even provided a version number to reference so all you know is that you are on the latest with a few notes from the developer on what you are getting. Finally, there is no cost to upgrade once you own an app, so the financial disincentive to upgrade that was the norm with paid desktop apps was entirely removed.

- Security built in
 - While security vulnerabilities are the number one reason to install a patch, security concerns is also the number one reason **not** to upgrade. Being the first to upgrade puts you at risk if a vulnerability in the new software is discovered, which is why corporate IT policies typically forbid end users from upgrading their desktop apps for a certain period of time. However, Apple fixed this by integrating a sandboxed model where the installed applications are limited in their ability to access data, contacts, photos, location, camera, and many other features without explicitly being given permission. This, combined with the rigorous app review process they instituted on store submissions for developers, reduced malware and app viruses to the point where security is not a concern for consumers when upgrading their apps.

Flow diagram showing consumer app update flow

Figure 8-14. User Model for iOS App Update Acceptance

The combination of simple upgrades that are low risk makes the decision to update very simple. This added with the fact releases are verified by a trusted authority means that users almost always make the decision to upgrade as shown in **Figure 8-14**.

The Apple App Store model is not ubiquitous for not only mobile devices, but also desktop application installation. Google took a similar model with their Android operating system in 2008, and both Apple and Microsoft introduced desktop app stores in 2011. Many of these app stores not only make it simple to upgrade to the latest version, but also offer an option to automatically upgrade.

As a result, client applications are now the norm on mobile devices and have seen a resurgence on desktop thanks to a few basic continuous update best practices:

CONTINUOUS UPDATE BEST PRACTICES

- Automatic Updates
 - Problem: Manual updates are often skipped or deferred due to risk or just time involved to evaluate risk.
 - Solution: The best type of update is one that requires user interaction and just happens automatically (and safely). If the update is low risk and trusted, this eliminates the decision on whether the features are worthwhile since it can happen without human intervention.
- Frequent Updates
 - Problem: Lots of small and low risk updates are better than a single large and infrequent one that is risky to end users.
 - Solution: Provide updates often and with small, low-risk changes. The store model encourages this by making small updates easier to get verified and released, and also by the OS showcasing updated apps which increases engagement.

Continuous Uptime

In the cloud age, one of the most important measures for business success is service uptime. Rather than just delivering software, many companies are moving to a software-as-a-service (SaaS) model where they are also responsible for the infrastructure the software runs on. Unexpected interruptions of service can be extremely costly, both in breach of service level agreements, but also in customer satisfaction and retention.

While uptime is important for all business provided internet services, there is no place that uptime is more important than in companies that build and support the very infrastructure the internet relies upon. Let's take a deeper look at one of the internet giants that runs global infrastructure underlying over 10% of the websites in the world and a myriad of applications and services that we rely upon on a daily basis.

Case Study: Cloudflare

As internet usage has exploded, so has the need for highly reliable, globally distributed, and centrally managed infrastructure like Content Delivery Networks (CDNs). Cloudflare's business is providing a highly reliable content delivery infrastructure to businesses across the world with the promise that they can deliver content faster and more reliably than your own infrastructure or cloud computing servers can. This also means that they have one job, which is **never** to go down.

While Cloudflare has had many production issues over the years involving DNS outages, buffer overflow data leaks, and security breaches, as their business has grown the scale of the problem and resulting damage has gone up. Five of these outages occurred on a global scale, taking out an increasingly large portion of the internet. While many of us are probably happy to have a 30 minute break from the internet courtesy of continuous update failures (after which we promptly write about them in our book), losing access to 100s of millions of servers across the internet can cause major disruptions for businesses and huge financial loss.

We are going to focus on the three most recent global Cloudflare outages, what

we are going to focus on the three most recent global Cloudflare outages, what happened, and how they could have been prevented with continuous update best practices.

2013 Cloudflare Router Rule Outage

CloudflarePartnerPalooza

Figure 8-15. Cloudflare at Partner Palooza in September 2013

In 2013 Cloudflare, shown in **Figure 8-15**, was at a scale where they had 23 data centers across 14 countries serving 785,000 websites and over 100 billion page views per month^{footnote}:[\[https://techcrunch.com/2013/03/03/cloudflare-is-down-due-to-dns-outage-taking-down-785000-websites-including-4chan-wikileaks-metallica-com/\]](https://techcrunch.com/2013/03/03/cloudflare-is-down-due-to-dns-outage-taking-down-785000-websites-including-4chan-wikileaks-metallica-com/). At 9:47 UTC on March 3rd Cloudflare had a system-wide outage affecting all of their data centers when they effectively dropped off the internet.

After the outage commenced, it took about 30 minutes to diagnose the problem and a full hour for all services to be restored at 10:49 UTC. The outage was caused by a bad rule that was deployed to the Juniper routers that sat on the edge of all their data centers shown in **Example 8-1**. It was intended to prevent an ongoing DDoS attack that had unusually large packets in the range of 99,971 to 99,985 bytes. Technically the packets would have been discarded after hitting the network since the largest allowed packet size was 4,470, but this rule was intended to stop the attack at the edge before it impacted other services.

Example 8-1. The rule that caused Cloudflare's routers to crash²⁵

```
+ route 173.X.X.X/32-DNS-DROP {  
+     match {  
+         destination 173.X.X.X/32;  
+         port 53;  
+         packet-length [ 99971 99985 ];  
+     }  
+     then discard;  
+ }
```

This rule caused the Juniper edge routers to consume all RAM until they crashed. Removing the offending rule fixed the problem, but many routers were in a state where they could not be automatically rebooted and required manual power cycling.

While Cloudflare blamed Juniper networks and their Flowspec system that deploys rules across a large cluster of routers, they are the ones who deployed an untested rule to their hardware with no ability to failover or rollback in the case of failure.

- Progressive Delivery
 - Problem: In a distributed system, deploying new code to all production nodes (in this case routers) simultaneously also breaks them all simultaneously.
 - Solution: Using canary releases you can deploy the change to a few nodes first and test for issues before continuing the update. If there is an issue simply rollback the affected nodes and debug it offline.
- Local Rollbacks
 - Problem: Reconfiguring edge devices can cause them to lose internet connectivity, making it difficult or impossible to reset them. In this case, an additional 30 minutes of downtime was spent manually resetting routers across 23 data centers and 14 countries.
 - Solution: Design edge devices to store the last known good configuration and restore to that in the case of an update failure. This preserves network connectivity for subsequent network fixes.

2019 Cloudflare Regex Outage

By 2019 Cloudflare had grown to host 16 million internet properties, serve 1 billion IP addresses, and in totality power 10% of the Fortune 1000 companies²⁶. They had a very good run of six years with no global outages until 13:42 UTC on July 2nd when Cloudflare-proxied domains started returning 502 “Bad Gateway” errors and remained down for 27 minutes.

This time the root cause was a bad regular expression, shown in **Example 8-2**. When this new rule was deployed to the Cloudflare Web Application Firewall (WAF) it caused the CPU usage to spike on all cores handling HTTP/HTTPS traffic worldwide.

Example 8-2. The regular expression that caused Cloudflare’s outage²⁷

```
(?: (?:\"|'|\\]|\\}|\\d|
```

```
(?:nan|infinity|true|false|null|undefined|symbol|math)|\\`|\\-|\\+|[+)]*)*;?  
((?:\\s|-|~|!|{|}|\\||\\\\|\\+)*\\.?(?:\\.*=.*))
```

Like any good regular expression, no human is capable of reading and understanding the series of unintelligible symbols, and certainly has no chance of verifying the correctness visually. In retrospect it is obvious that the buggy part of the Regex is `.\ *(?:.\ *=.\ *)`. Since part of this is a non-capturing group, for the purposes of this bug it can be simplified to `.*.\ *=.*`. The use of a double, non-optional wildcard (```.*```) is known to be performance issue with regular expressions since they must perform backtracking that gets super-linearly harder as the length of the input to be matched increases. If you made it this far through the explanation... you are clearly on the genius end of **Figure 8-16**.

yesterdays regex

Figure 8-16. Yesterday's Regex comic from Geek&Poke²⁸

Given the difficulty of manually verifying bugs that get deployed to global infrastructure, you would think that they would have learned from their 2013 outage and implemented progressive delivery. In fact, they had since implemented a very complex progressive delivery system that involved three stages:

- **DOG Point-of-Presence:** The first line of defense on new changes used only by Cloudflare employees. Changes get deployed here first so issues can be detected by employees before it gets into the real world.
- **PIG Point-of-Presence:** A Cloudflare environment where a small subset of customer traffic goes and new code can be tested without affecting paying customers.
- **Canary Point-of-Presence:** Three global canary environments that get a subset of worldwide traffic as a last line of defense before changes go global.

Unfortunately, the Web Application Firewall was primarily used for fast threat response, and as a result it bypassed all of these canary environments and went straight to production. In this case the regular expression was only run through a series of unit tests that did not check for CPU exhaustion before it was pushed to production. This particular change was not an emergency fix and thus could have done a staged rollout following the above process.

The exact timeline of the problem and subsequent fix was as follows:

- 13:31 - Code check-in of the peer reviewed Regex
- 13:37 - CI server built the code and ran the tests, which passed
- 13:42 - The erroneous Regex was deployed to the WAF in production
- 14:00 - The possibility of an attacker was dismissed and the WAF was identified as the root cause
- 14:02 - It was decided to go a global WAF kill
- 14:07 - The kill was finally executed after delays accessing the internal

systems

- 14:09 - Service restored for customers

To close this out, let's review the continuous update best practices that may have helped Cloudflare avoid another global outage:

CONTINUOUS UPDATE BEST PRACTICES

- Progressive Delivery
 - Problem: Canary deployments had now been implemented, but was not used for the Web Application Firewall (WAF) since this was used for fast threat response.
 - Solution: This particular rule was no an emergency and could have followed the canary deployment process.
- Observability
 - Problem: Some problems are hard to trace relying on user feedback only.
 - Solution: Implement tracing, monitoring, and logging in production. Cloudflare actually had specifically implemented a production watchdog designed to prevent excessive CPU use by regular expressions that would have caught this issue. However, a few weeks prior this code was removed to optimize the WAF to use less CPU.

2020 Cloudflare Backbone Outage

Image of Discord client displaying Cloudflare outage notification



Figure 8-17. Discord pointing their users to the upstream outage on Cloudflare

12 months after the previous Cloudflare outage, I am sitting down to write about their 2019 outage and two very peculiar things happened:

- Around 2:12PM PST (21:12 UTC) the family Discord channel stops going off because of [Figure 8-17](#) and I become incredibly productive.
- A few hours later all my searches for information on Cloudflare outages start turning up information on recent DNS issues instead of the articles from last year.

The nice folks at Cloudflare clearly recognized that good case studies come in threes and provided another antipattern for this chapter. On July 18th, 2020, Cloudflare had another production outage for 27 minutes that affected 50% of their total network^{[29](#)}.

This time the issue was with the Cloudflare's backbone, which is used to route the majority of the traffic on their network between major geographies. To understand how the backbone works, it helps to understand the topology of the internet. The internet is not truly point-to-point but instead relies on a complex network of interconnected data centers to transmit information as shown in [Figure 8-18](#).

Network visualization of internet traffic

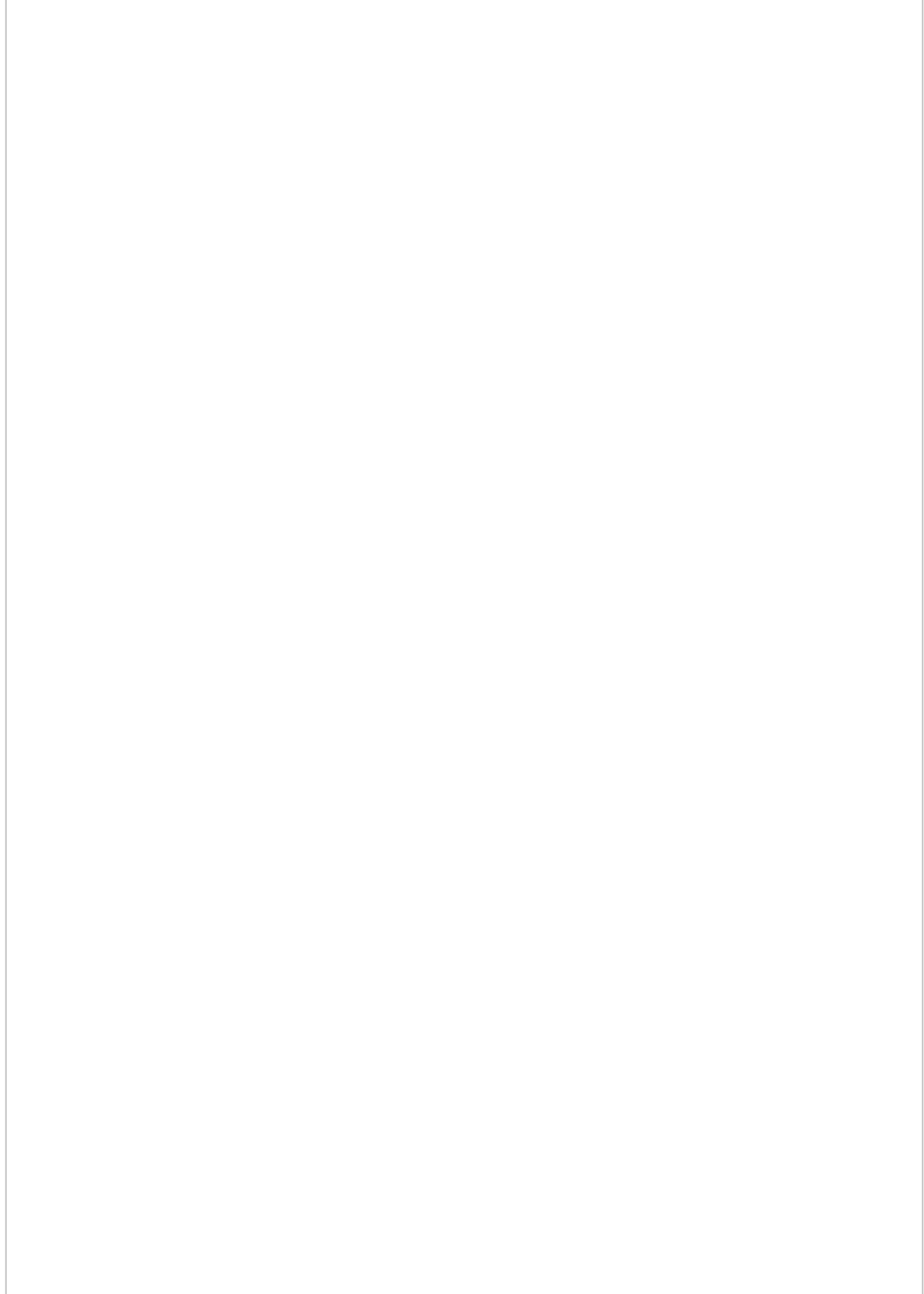


Figure 8-18. Global map of internet interconnects³⁰

The dots where many points converge are network data centers like the ones that Cloudflare runs in San Jose, Atlanta, Frankfurt, Paris, São Paulo, and other cities worldwide. These data centers are connected by a global backbone of direct, high-speed connections that allow them to bypass internet congestion and improve quality of service between major markets.

It is the Cloudflare backbone which was the cause of the outage this time. The backbone is designed to be resilient to failures, such as the one that happened between Newark and Chicago at 20:25 UTC. However, this outage resulted in increased congestion between Atlanta and Washinton, D.C.. The attempted fix was to remove some of the traffic from Atlanta by executing the routing change in [Example 8-3](#).

Example 8-3. The routing change that caused Cloudflare's network to go down

```
{master}[edit]
atl01# show | compare
[edit policy-options policy-statement 6-BBONE-OUT term 6-SITE-LOCAL
from]
!      inactive: prefix-list 6-SITE-LOCAL { ... }
```

This routing change inactivates one line of a term script, which is shown in [Example 8-4](#).

Example 8-4. The complete term the change was made on

```
from {
    prefix-list 6-SITE-LOCAL;
}
then {
    local-preference 200;
    community add SITE-LOCAL-ROUTE;
    community add ATL01;
    community add NORTH-AMERICA;
    accept;
}
```

The correct change would have been to inactivate the entire term. However, by removing the `prefix-list` line the result was to sent this route to all other backbone routers. This changed the `local-preference` to 200, which gave Atlanta priority over the other routes, which were set to 100. The result was that rather than reducing traffic, Atlanta instead started attracting traffic from across

the backbone, increasing network congestion to the point where internet was disrupted for half of Cloudflare's network.

There is a lot to say about configuration changes that can destroy your entire business. The core of the problem here is that Cloudflare is not treating the configuration of backbone routers as code that is properly peer reviewed, unit tested, and canary deployed.

CONTINUOUS UPDATE BEST PRACTICES

- Automated Testing
 - Problem: If you don't have automation around your code and configuration then manual errors will go into production uncaught.
 - Solution: All code (including configuration) needs to be tested in an automated fashion so it is repeatable.
- Observability
 - Problem: Detecting errors through observed behavior is error prone and time consuming when seconds and minutes spell disaster. It took Cloudflare 27 minutes to identify there was a problem and trace it back to the Atlanta router.
 - Solution: Implement tracing, monitoring, and logging in production, especially for your key assets like the internet backbone.

The Hidden Cost of Manual Updates

Implementing continuous update best practices is not free, and often it can seem more cost effective to delay automation and continual manual processes. In particular, doing automated testing, treating configuration like code, and automating deployment are all important, costly.

However, what is the hidden cost of not automating your deployment? Manual

deployments are fraught with errors and mistakes that cost time and effort to troubleshoot and business loss when they negatively impact customers. What is the cost of having production errors that persist for hours as staff is brought in to troubleshoot the issue on a live system?

In the case of Knight Capital, where the answer turned out to be 10 million dollars per minute of system failure, would you trust manual updates?

Case Study: Knight Capital

Knight Capital is an extreme case of a software bug going undetected, causing issues in production, and costing a huge amount of financial loss. However, the interesting thing about this bug is that the core issue was mistakes made in the deployment process, which was both infrequent and manual. If Knight Capital was practicing continuous deployment they would have avoided a mistake which ended up costing them 440 million and control of the company.

Knight Capital was a market making trader specializing in high volume transactions and throughout 2011 and 2012 their trading in U.S. equity securities represented approximately 10 percent of the market volume. They had several internal systems that handled trade processing, one of which was called Smart Market Access Routing System (SMARS). SMARS acted as a broker, taking trading requests from other internal systems and executing them in the market³¹.

To support a new Retail Liquidity Program (RLP) that was set to launch on August 1st of 2012, Knight Capital upgraded their SMARS system to add in new trading functionality. They decided to reuse the API flag for a deprecated function called Power Peg that was meant for internal testing only. This change was thought to have been successfully deployed to all eight production servers in the week leading up to the RLP launch.

At 8:01 AM EST the morning of August 1st started with some suspicious, but sadly ignored, e-mail warnings about errors on pre-market trading orders that referenced SMARS and warned “Power Peg disabled”. Once trading commenced at 9:30 AM EST SMARS immediately started executing a large volume of suspicious trades that would repeatedly buy high (at offer) and sell low (at bid), immediately losing on the spread. Millions of these transactions were being queued at 10ms intervals, so even though the amounts were small (15

cents on every pair of trades), the losses piled up extremely quickly³².

Expression of a Knight Capital trader captured on camera.

Figure 8-19. Expression of Jason Blatt on the trading floor from Knight Capital³³

In a business where seconds can be costly, minutes can wipe out weeks of earnings, and an hour is a lifetime, Knight Capital lacked an emergency response plan. During this 45m period they executed 4 million orders for trading 397 million shares of 154 stocks³⁴. This gave them a net long position of 3.4 billion and a net short position of 3.15 billion. After getting 6 of the 154 stocks reversed and selling the remaining positions they were left with a net loss of around 468 million. This was a very tough period for Knight Capital as captured from Jason Blatt's expression on the trading floor as shown in **Figure 8-19**.

Backtracking to the root cause of this problem, only seven of the eight production servers were correctly upgraded with the new RLP code. The last server had the old Power Peg logic enabled on the same API flag, which explains the warning e-mails earlier in the morning. For every request which hit this eighth server an algorithm designed for internal testing was run that executed millions of inefficient trades designed to quickly bump the price of the stock up. However, in troubleshooting this problem the technical team erroneously thought that there was a bug on the newly deployed RLP logic and reverted back the code on the other seven servers, essentially breaking 100% of the transactions and exacerbating the problem.

While Knight Capital did not go entirely bankrupt from this, they had to give up 70% of control of the company for a 400 million bailout out their position. Before the end of the year this turned into an acquisition by one of their competitors, Getco LLC, and the resignation of CEO Thomas Joyce.

So, what happened to Knight Capital and how can you avoid a disaster like this? Let's cover some additional continuous update best practices:

CONTINUOUS UPDATE BEST PRACTICES

- Frequent Updates
 - Problem: If you only occasionally update your system, chances are that you are not very good (or efficient) at this and mistakes will be made. Knight Capital did not have the proper automation or control checks in place to reliably

make production changes and were a continuous update disaster waiting to happen.

- Solution: Update often and automate always. This will build the organizational muscle memory so that simple updates become routine and complex updates are safe.
- State Awareness
 - Problem: The target state can affect the update process (and any subsequent rollbacks).
 - Solution: Know and consider target state when updating. Reverting might require reverting the state.

Continuous Update Best Practices

Now that you have seen the dangers of not adoption continuous update best practices from a variety of companies in different areas of the technology industry, it should be obvious why you should start implementing or continue to improve your continuous deployment infrastructure.

The following is a list of all the continuous update best practices along with the case studies that go into them in more detail:

- Frequent Updates
 - The only way to get good at updating is to do it a lot
 - Case Studies: iOS App Store, Knight Capital
- Automatic Updates
 - And if you are updating a lot it becomes cheaper and less error prone to automate
 - Case Studies: iOS App Store
- Automated Testing

- The only way to make sure you are deploying quality is to test everything on every change
- Case Studies: Java Six Month Release Cadence, 2020 Cloudflare Backbone Outage
- Progressive Delivery
 - Avoid catastrophic failures by deploying to a small subset of production with a rollback plan
 - Case Studies: 2013 Cloudflare Router Rule Outage, 2019 Cloudflare Regex Outage
- State Awareness
 - Don't assume that code is the only thing that needs to be tested, state exists and can wreak havoc in production
 - Case Studies: Knight Capital
- Observability
 - Don't let your customers be the ones to notify you that you are down!
 - Case Studies: 2019 Cloudflare Regex Outage, 2020 Cloudflare Backbone Outage
- Local Rollbacks
 - Edge devices are typically numerous and hard to fix once a bad update hits, so always design for a local rollback
 - Case Studies: 2013 Cloudflare Router Rule Outage

Now that you are armed with knowledge, it is time to start convincing your coworkers to adopt best practices today, before you become the next Knight Capital “Knightmare” of the high tech industry. Making headlines is great, but do it as an elite performer of the DevOps industry rather than on the front page of the Register.

-
- 1 “Accelerate: State of DevOps 2019”, 2019. <https://services.google.com/fh/files/misc/state-of-devops-2019.pdf>
 - 2 © Raimond Spekking / CC BY-SA 4.0 (via Wikimedia Commons)
(https://commons.wikimedia.org/wiki/File:Nokia_6110_blue-92107.jpg),
<https://creativecommons.org/licenses/by-sa/4.0/legalcode>
 - 3 Public Domain, <https://www.dvidshub.net/image/6184831/mc20-oil-platform-prior-collapse>
 - 4 “An Integrated Assessment of Oil and Gas Release into the Marine Environment at the Former Taylor Energy MC20 Site”. National Centers for Coastal Ocean Science (NCCOS/NOAA). June 1, 2019. <https://coastalscience.noaa.gov/project/mc20report/>
 - 5 Tim Donaghy (February 19, 2016). “This Oil Leak Could Last for 100 Years — and the Company Involved Refuses to Fix It”. Greenpeace. Retrieved October 23, 2018.
 - 6 2020 Open Source Security and Risk Analysis (OSSRA) Report,
<https://www.synopsys.com/content/dam/synopsys/sig-assets/reports/2020-ossra-report.pdf>
 - 7 Public Domain, <https://www.dvidshub.net/image/6184832/mc20-oil-platform-after-collapse-with-containment>
 - 8 Public Domain, <https://securelist.com/wannacry-ransomware-used-in-widespread-attacks-all-over-the-world/78351/>
 - 9 Ungood-Thomas, Jon; Henry, Robin; Gadher, Dipesh (14 May 2017). “Cyber-attack guides promoted on YouTube”. The Sunday Times.
 - 10 Ptrump16 (https://commons.wikimedia.org/wiki/File:Siemens_Magnetom_Aera_MRI_scanner.jpg),
<https://creativecommons.org/licenses/by-sa/4.0/legalcode>
 - 11 Comptroller and Auditor General (2018). “Investigation: WannaCry cyber attack and the NHS”. National Audit Service., <https://www.nao.org.uk/wp-content/uploads/2017/10/Investigation-WannaCry-cyber-attack-and-the-NHS.pdf>
 - 12 <https://www.fbi.gov/wanted/cyber/chinese-pla-members-54th-research-institute>
 - 13 <https://www.bankinfosecurity.com/equifax-data-breach-costs-hit-14-billion-a-12473>
 - 14 <https://nvd.nist.gov/vuln/detail/CVE-2017-5638>
 - 15 Guy Bruneau, March 25th 2018. “Scanning for Apache Struts Vulnerability CVE-2017-5638”
<https://isc.sans.edu/forums/diary/Scanning+for+Apache+Struts+Vulnerability+CVE20175638/23479/>
 - 16 Jann Horn, Jan 3 2018. “Reading privileged memory with a side-channel”
<https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html>
 - 17 Anton Shilov, October 8, 2018. “Intel’s New Core and Xeon W-3175X Processors: Spectre and Meltdown Security Update” <https://www.anandtech.com/show/13450/intels-new-core-and-xeon-w-processors-fixes-for-spectre-meltdown>
 - 18 “Java version history”, https://en.wikipedia.org/wiki/Java_version_history
 - 19 Snyk, “JVM Ecosystem Report 2020”, 2020. https://snyk.io/wp-content/uploads/jvm_2020.pdf
 - 20 Stephen Coulbourn, “Java is still available at zero-cost”, 2018. <https://blog.joda.org/2018/08/java->

is-still-available-at-zero-cost.html

- 21 “Java 8 Adoption in March 2016”, <https://www.baeldung.com/java-8-adoption-march-2016>
- 22 <https://mreinhold.org/blog/modular-java-platform>
- 23 <https://en.wikipedia.org/wiki/WorldWideWeb>
- 24 Apple Inc.
([https://commons.wikimedia.org/wiki/File:Available_on_the_App_Store_\(black\)_SVG.svg](https://commons.wikimedia.org/wiki/File:Available_on_the_App_Store_(black)_SVG.svg)),
„Available on the App Store (black) SVG“, marked as public domain, more details on Wikimedia
Commons: <https://commons.wikimedia.org/wiki/Template:PD-text>
- 25 <https://blog.cloudflare.com/todays-outage-post-mortem-82515/>
- 26 <https://thehill.com/policy/cybersecurity/451331-websites-go-down-worldwide-due-to-cloudflare-outage>
- 27 <https://blog.cloudflare.com/details-of-the-cloudflare-outage-on-july-2-2019/>
- 28 Oliver Widder, Geek&Poke, “Yesterdays Regex”, Dec. 2013. CC-BY-3.0, <http://geek-and-poke.com/geekandpoke/2013/12/3/yesterdays-regex>
- 29 <https://blog.cloudflare.com/cloudflare-outage-on-july-17-2020/>
- 30 The Opte Project (https://commons.wikimedia.org/wiki/File:Internet_map_1024.jpg), „Internet map 1024“, <https://creativecommons.org/licenses/by/2.5/legalcode>
- 31 <https://medium.com/dataseries/the-rise-and-fall-of-knight-capital-buy-high-sell-low-rinse-and-repeat-ae17fae780f6>
- 32 <https://www.zerohedge.com/news/what-happens-when-hft-algo-goes-totally-berserk-and-serves-knight-capital-bill>
- 33 <https://www.gettyimages.com/detail/news-photo/jason-blatt-of-knight-capital-americas-lp-reacts-to-down-news-photo/153652001>
- 34 <https://www.sec.gov/litigation/admin/2013/34-70694.pdf>

About the Authors

Stephen Chin is Head of Developer Relations at JFrog and author of The Definitive Guide to Modern Client Development, Raspberry Pi with Java, and Pro JavaFX Platform. He has keynoted numerous Java conferences around the world including Devvxx, JNation, JavaOne, Joker, and Open Source India. Stephen is an avid motorcyclist who has done evangelism tours in Europe, Japan, and Brazil, interviewing hackers in their natural habitat. When he is not traveling, he enjoys teaching kids how to do embedded and robot programming together with his teenage daughter. You can follow his hacking adventures at: <http://steveonjava.com>

Melissa McKay is currently a Developer Advocate with the JFrog Developer Relations team. She has been active in the software industry 20 years and her background and experience spans a slew of technologies and tools used in the development and operation of enterprise products and services. Melissa is a mom, software developer, Java geek, huge promoter of Java UNconferences, and is always on the lookout for ways to grow, learn, and improve development processes. She is active in the developer community, has spoken at CodeOne, Java Dev Day Mexico and assists with organizing the JCrete and JAlba Unconferences as well as Devvxx4Kids events.

Ixchel Ruiz has developed software applications and tools since 2000. Her research interests include Java, dynamic languages, client-side technologies, and testing. She is a Java Champion, Groundbreaker Ambassador, Hackergarten enthusiast, open source advocate, JUG leader, public speaker, and mentor.

Baruch Sadogursky (a.k.a. JBaruch) is the Chief Sticker Officer (also, Head of DevOps Advocacy) at JFrog. His passion is speaking about technology. Well, speaking in general, but doing it about technology makes him look smart, and 19 years of hi-tech experience sure helps. When he's not on stage (or on a plane to get there), he learns about technology, people and how they work, or more precisely, don't work together.

He is a co-author of the Liquid Software book, a CNCF ambassador and a passionate conference speaker on DevOps, DevSecOps, digital transformation, containers and cloud-native, artifact management and other topics, and is a regular at the industry's most prestigious events including DockerCon, Devvxx, DevOps Days, OSCON, Qcon, JavaOne and many others. You can see some of

his talks at jfrog.com/shownotes

1. 1. DevOps for (or Possibly Against) Developers

- a. DevOps is an entirely invented concept, and the inventors came from the Ops side of the equation
 - i. Exhibit #1: The Phoenix Project
 - ii. Exhibit #2: The DevOps Handbook
 - iii. Google It
 - iv. What Does It Do?
 - v. State of the Industry
 - vi. What Constitutes Work?
- b. If We're Not About Deployment and Operations, Then Just What Is Our Job?
 - i. Just What Does Constitute Done?
 - ii. Rivalry?
- c. More Than Ever Before
 - i. Volume and Velocity
 - ii. Done and Done
 - iii. Fly Like a Butterfly...
 - iv. Integrity, Authentication, and Availability
 - v. Fierce Urgency
- d. The software industry has fully embraced DevOps
 - i. Making It Manifest
 - ii. We all got the message

2. 2. The System of Truth

- a. Three Generations of Source Code Management
 - b. Choosing Your Source Control
 - c. Making Your First Pull Request
 - d. Git Tools
 - i. Git Command Line Basics
 - ii. Git Command Line Tutorial
 - iii. Git Clients
 - iv. Git IDE Integration
 - e. Git Collaboration Patterns
 - i. git-flow
 - ii. GitHub Flow
 - iii. Gitlab Flow
 - iv. OneFlow
 - v. Trunk Based Development
 - f. Summary
3. 3. Dissecting the Monolith
- a. Monolithic architecture
 - i. Granularity and functional specification
 - ii. Cloud Computing
 - iii. Service Models
 - b. Microservices
 - i. Definition
 - ii. Anti-Patterns

- iii. DevOps & Microservices
 - iv. Microservice Frameworks
 - v. Spring Boot
 - vi. Micronaut
 - vii. Quarkus
 - viii. Helidon
 - c. Serverless
 - i. Setting Up
 - d. Conclusion
- 4. 4. Continuous Integration
 - a. Adopt Continuous Integration
 - b. Declaratively Script Your Build
 - i. Build With Apache Ant
 - ii. Build With Apache Maven
 - iii. Build With Gradle
 - c. Automate Tests
 - i. Run Unit Tests Automatically
 - ii. Monitor and Maintain Tests
 - iii. Speed Up Your Test Suite
 - d. Continuously Build
- 5. 5. Package Management
 - a. Why build-it-and-ship-it is not enough
 - b. It's all about metadata

- i. What's metadata?
 - ii. Determining the metadata
 - iii. Capturing metadata
 - iv. Writing the metadata
 - c. Dependency management basics for Apache Maven & Gradle
 - i. Dependency management with Apache Maven
 - ii. Dependency management with Gradle
 - d. Dependency management basics for containers
 - e. Artifact Publication
 - i. Publishing to Maven Local
 - ii. Publishing to Maven Central
 - iii. Publishing to Sonatype Nexus
 - iv. Publishing to JFrog Artifactory

6. 6. Securing Your Binaries

- a. Supply Chain Security Compromised
 - i. What Happened at SolarWinds?
 - ii. Security from the Vendor Perspective
 - iii. Security from the Customer Perspective
 - iv. The Full Impact Graph
- b. Securing your DevOps infrastructure
 - i. The Rise of DevSecOps
 - ii. The Role of SREs in Security
- c. Static and Dynamic Security Analysis

- i. Static Application Security Testing
 - ii. Disadvantages of the SAST approach
 - iii. Dynamic Application Security Testing
 - iv. Comparing SAST and DAST
- d. The Common Vulnerability Scoring System
 - i. CVSS Basic Metrics
 - ii. CVSS Temporal Metrics
 - iii. CVSS Environmental Metrics
 - iv. CVSS Summary
- e. Extent of Security Scanning
 - i. Time to Market
 - ii. Make or Buy
 - iii. One-time and Recurring Efforts
 - iv. How much is enough?
- f. Compliance versus Vulnerabilities
 - i. Compliance Issues: Singular Points in your Full-Stack
 - ii. Vulnerabilities: Can be Combined into Different Attack-Vectors
 - iii. Vulnerabilities: Timeline from Inception Through Production Fix
 - iv. Test Coverage is your Safety-Belt
- g. Security scanning as a Promotion Quality Gate
 - i. Fit with Project Management Procedures
 - ii. Implementing Security with the Quality Gate Method

- iii. Risk Management in Quality Gates
 - iv. Practical Applications of Quality Management
- h. Shift Left to the CI and the IDE
 - i. Not All Clean Code is Secure Code
 - ii. Effects on Scheduling
 - iii. The Right Contact Person
 - iv. Dealing with Technical Debt
 - v. Advanced Training on Secure Coding
 - vi. Milestones for Quality
 - vii. The Attacker's Point of View
 - viii. Methods of Evaluation
 - ix. Be Aware of Responsibility

7. 7. Mobile Workflows

- a. Fast-paced DevOps workflows for mobile
- b. Android Device Fragmentation
 - i. Android OS Fragmentation
 - ii. Building for Disparate Screens
 - iii. Hardware and 3D Support
- c. Continuous Testing on Parallel Devices
 - i. Building a Device Farm
 - ii. Mobile Pipelines in the Cloud
 - iii. Planning a Device Testing Strategy
- d. Summary

- 8. 8. Continuous Deployment Patterns and Antipatterns
 - a. Why Everyone Needs Continuous Updates
 - i. User Expectations on Continuous Updates
 - ii. Security Vulnerabilities Are the New Oil Spills
 - b. Getting Users to Update
 - i. Case Study: Java Six Month Release Cadence
 - ii. Case Study: iOS App Store
 - c. Continuous Uptime
 - i. Case Study: Cloudflare
 - d. The Hidden Cost of Manual Updates
 - i. Case Study: Knight Capital
 - e. Continuous Update Best Practices