CI/CD Pipeline with Docker and Jenkins

Learn How to Build and Manage Your CI/CD Pipelines Effectively

Sandeep Rawat

bpb

CI/CD Pipeline with Docker and Jenkins

Learn How to Build and Manage Your CI/CD Pipelines Effectively

Sandeep Rawat

bpb

CI/CD Pipeline with Docker and Jenkins

Learn How to Build and Manage Your CI/CD Pipelines Effectively

Sandeep Rawat

www.bpbonline.com

First published: 2023

Published by BPB Online

WeWork

119 Marylebone Road

London NW1 5PU

UK | UAE | INDIA | SINGAPORE

ISBN 978-93-55513-502

www.bpbonline.com

Dedicated to

*To the late **Mr. Dalbir Singh Rawat**, my uncle (Chacha)*

Remembering a life well led and all the joy it brought to so many.

In memory of a truly amazing person

About the Author

Sandeep has 18+ years of extensive experience in application development and creating a DevOps ecosystem with end-to-end automation. He is a firm believer in the evangelism of technology, and also, he believes in giving back to society. He makes a lot of open-source contributions to different problems in the technology world.

He likes to share his knowledge through well-read blogs, well-attended training, and public workshops. He has streamlined DevOps discipline and practices for different organizations.

Currently, Sandeep is running 3 organizations under TechPrimo Solutions. Opstree Solutions is a highly specialized Cloud and DevSecOps engineering company. The Technology Transformation Partner that specializes in making the application delivery lean, nimble and highly productive through the best in breed Cloud (Public, Private/Hybrid) and DevOpSecOps platform implementations, that enable getting to market faster, more securely, and with the ability to maintain the platform much more efficiently. OpsTree Labs the product arm, BuildPiper, a product of OpsTree Solutions, is an end-to-end Kubernetes and Microservices Application Delivery Platform that enables production-grade Microservices management for seamless Day 0, 1 and 2 operations and makes Kubernetes Microservices Application ready! MyGurukulam is a tech - centric upskilling platform aiming to bridge the skill set gap.

About the Reviewer

Kumar Gejara is a cloud architect living in Austin Texas, US, working for Apex Systems and managing the DevOps Centre of Excellence. He has 23 years of IT experience building integration and cloud solutions and delivering CI/CD platforms. His expertise includes cloud migration, monitoring, governance, automation, Infrastructure as Code (IaC), Kubernetes, machine learning operations (MLOps), DevOps, DevSecOps, DataOps, and cloud cost optimization.

He worked with a few start-ups such as Aalyance (acquired by HCL) and Tescra early in his career. Later, he also worked for companies such as Wipro, Perficient Inc, Insight Global and Apex Systems, and has had the opportunity to serve over 20 end clients since 2000. Kumar completed his Master Of Computer Applications (MCA, 1999) and Bachelor of Science(BSc, 1996) at Sri Venkateswara University, India.

You can contact him on his LinkedIn Profile: **https://www.linkedin.com/in/kumar-gejara-b621285/**

Acknowledgement

I have to start by thanking my family.

DS Rawat, my father, for always being an ideal near-perfect personality. I can only dream of being like him and inheriting his conscientiousness.

Kanta Rawat, my mother, for being a silent force and ensuring that we get all the support and freedom to become what we are in our family.

Sonal, my wife, for always being a rock-solid supporter of all of my endeavors, whether running a company (24x7) or writing a book. I never felt the pressure of de-prioritizing these endeavors for my family.

Jai and Yami, the two invaluable god gifts of my life, for keeping me recharged with a single word "Papa," and rejuvenating me with their hugs and smiles.

Next comes Abhishek, Adeel, and Sajal. I can't imagine the quality or completion of this book without the involvement of these three souls, Adeel being the creative writer and Abhishek and Sajal being the tech brain behind the book. I can only say that they are vital contributors to defining their role.

Now, the gratitude goes to OpsTree and BuildPiper, companies I founded, and my friend/mentor/guiding light, Shankar Jha. Through these companies, we came across a plethora of problems and solutions, a lot of which contributed to the contents of this book. Especially BuildPiper, which focused exclusively on solving CI-CD problems for our clients, allowing us to dig toward great depths of understanding. Since our agenda was to streamline CI-CD for any or all possibilities, we went through countless iterations of problem-solving and identified the best approaches to do the job.

Without further ado, I’d like to thank my second family, the people of Opstree. Meeting and engaging with hundreds of them daily regarding their projects, problems, and views made it possible for me to gather enough information to write a book. Their contributions cannot be overstated.

Many thanks also go to our clients, who contributed vastly to the sample space of our scenarios. They presented us with unique challenges enabling us to think in various directions. Even the repeated cases continuously improved

our processes and made them robust.

I’d like to thank the excellent team of BPB as well for always being available, reviewing, suggesting changes, and ensuring the book's quality remains impeccable.

One last name on the list of thanks is my favorite book, the Phoenix Project. When the offer to write this book came, I spent a lot of time thinking about what my book's writing style should be. What will get the complex concepts across and yet be engaging to read? I had to choose between the way of writing “The Phoenix Project” or HeadFirst. Ultimately, I decided to go ahead with the writing style of The Phoenix Project.

Preface

This book covers many aspects of Continuous Integration, Continuous Deployments, and their various roles and integration. This book also talks about the limitation of the SDLC world and how Docker and Continuous Integration can collaboratively solve those problems. It shows how we can create an ideal Continuous Integration pipeline with different integration checks like managing quality, testing, and security of the application codebase. Also, this book gives the importance of making the system more stable and resilient.

This book takes a practical approach where we use an actual application to show the journey of Continuous Integration. Also, this book covers information such as how to containerize an application with an effective way of writing a Dockerfile. This book also gives importance to Continuous Integration steps like- Continuous Deployment. You can use this book to understand the different technical environments and deployment strategies and what are the parameters to identify a deployment strategy for your environment.

This book is divided into eight chapters. They will cover the basics of SDLC, Continuous Integration, Docker and its essential concepts, and Continuous Deployment. So, learners will understand the complete journey of the modern SDLC implementation.

Chapter one, explains the book's methodology and the characters. The characters will discuss the problems in their SDLC and how continuous integration can solve them. Also, the project details and description will be part of it.

Chapter two, focuses on the issues and gaps between the Development and DevOps team. Also, in this chapter, the concept, and importance of Continuous Integration will be introduced. We will also discuss the integration checks of Continuous Integration.

Chapter three, explains the tooling landscape in which different Continuous Integration tools will be compared, and finally, Jenkins gets decided as the automation tool. Also, this chapter will talk about Jenkins and its essential

features, along with other integration tools.

Chapter four, shows the implementation of integration checks manually and automating them with Jenkins. Also, we will explain the concept of the Jenkins pipeline and how we can create the stages of integration checks in the pipeline.

Chapter five, discusses the common problem of the DevOps world, "Environment inconsistency." Also, we will discuss the possible solution to overcome the environmental inconsistency issues and how Docker and containerization can solve all these issues. Also, we will discuss Docker and its components.

Chapter six, shows how the existing Continuous Integration pipeline will be modified to cover the integration checks related to Docker and its stages. Also, we will explain the advantages and benefits of introducing Docker into the existing system.

Chapter seven, explains the role of Continuous Deployment, Continuous Delivery, and their importance after a successful Continuous Integration setup. Also, we will discuss and compare the different deployment strategies and how to choose an ideal design for your environment.

Chapter eight, shows the implementation of Continuous Deployment with different deployment strategies in different environments. Also, we will discuss the organization's technical environments and their purpose.

Code Bundle and Coloured Images

Please follow the link to download the
Code Bundle and the ***Coloured Images*** of the book:

https://rebrand.ly/xj6ofj1

The code bundle for the book is also hosted on GitHub at **https://github.com/bpbpublications/CI-CD-Pipeline-with-Docker-and-Jenkins**. In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at **https://github.com/bpbpublications**. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

eBook version at **www.bpbonline.com** and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at: **business@bpbonline.com** for more details.

At **www.bpbonline.com**, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks.

Piracy

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at **business@bpbonline.com** with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit **www.bpbonline.com**. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit **www.bpbonline.com**.

Table of Contents

Index

CHAPTER 1

Introduction

The major issue for any project is how to easily and safely release the new version of code. We have seen people struggling and getting frustrated working for hours to deal with deployment issues. The more they stray from standardized methodologies, the messier their release cycle becomes. Seeing the same problem project after the project moved us to write this book. The idea behind writing this book is simple: make releases as smooth as possible. This book aims to be the go-to solution for the implementation of CI/CD for every form of technology by focusing on why to opt for CI and how to apply it the modern way. In other words, we want our readers to need nothing beyond this book when seeking help related to any CI/CD topic.

The first step toward these goals, we thought, would be the flow, i.e., how we introduce topics and how we transition from one to another. This is the reason we chose to go with a writing style that is easiest to follow: a narrative style. We are inspired by '**The Phoenix Project** by Gene Kim, Kevin Behr, and George Spafford. Reading it is like reading a story, with its own twists and turns. At the end, you have not only understood complex concepts but also read a good story. It's effortless and apt for what we have in mind. It is very different from typical informative books that are not everyone's cup of tea. For those who haven't read '**The Phoenix Project**, we recommend you do.

Similarly, we would like to take our readers on a journey where the challenge would be improving the efficiency of the project by applying the principles of CI/CD with the help of Docker and Jenkins as the preferred technologies.

Structure

In this chapter, we will discuss the following topics:

- Character Introduction

- Sprint 1 - Retrospection
- Light of Hope

Objectives

The main objective of this chapter is to provide a detailed understanding of the concepts necessary for software development and how releasing a new version of software can become a pain for many organizations. You will be introduced to many characters, who are going to be important in the overall development and journey of applications.

There are a few additional things that you will learn from this chapter, like the concepts of sprint and sprint planning. This chapter will give an overall idea of the concepts and principles of continuous integration.

Character Introduction

A new day and a fresh morning. As I looked outside my window, I saw the world moving just like last week, unchanged, unlike me. The reason being interruption; I can't wait for a whole week to finish my ongoing thriller web series that grasped me till quite late last night. The problem with binge-watching is that it doesn't stop time. Sooner or later, you are forced to let go and worry about the next day. But today is not so bad; in fact, the last few days have been a little exciting as I am in the middle of my first month at the office, a new start with my current organization. Therefore, I am hurtling through my stuff right now, with the goal of reaching office in time. During my interview here, I had a good chat with my now reporting manager, Sandeep, about the goals and requirements of the company. That conversation was one of the major role players in my decision to join. These people are doing some good work, which makes way for ample opportunities for career growth. I can contribute in many ways. Even while discussing, I was full of ideas. Now that I am slowly nearing the end of all the onboarding formalities and KT sessions, I can't help but get excited. In an hour, as I geared myself up to beat the Monday blues, I kept thinking of the hint that Sandeep had given about a new assignment I would be working on. Minutes ago, I got an invite for a meeting to be held first thing in the next morning.

"Today's a sunny day.", I thought to myself as the car switched from the under-bridge to the over-bridge. "It's a nice day to face some new

challenges.” Since I am new to this company, I can’t help feeling an urge to prove my worth. I wonder if it’s only me. I bet people hired higher up in the hierarchy don’t care as much. Experience amounts to a lot. My thoughts continued as I neared the office building. After a pleasant trip on a sunny morning, I reached the office with endless thoughts revolving in my mind, about the exposure and challenges this new assignment would bring my way. After having a cup of coffee and sharing last weekend’s experiences with my new colleagues and friends, I walked up to the conference room 10 minutes prior to the scheduled time. At this point, I was hoping to get comfortable with the other new faces. In my experience, a formal introduction goes much better if you’ve had an informal one before. I feel more at ease and can crack better jokes. Hail Chandler! No one entered the room for the next 5 minutes. I was getting anxious as the butterflies started a brawl in my stomach. I kept looking at my phone, and just to be sure I hadn’t missed anything important, I quickly rolled down the notification dropdown which, I know, would go untouched, at least for the next hour. In the next 2-3 minutes, I was joined by two more brains. Thinking this to be my moment to shine, I got up and shook their hands enquiring, rather nonchalantly, about them. I didn’t want to give away my “not so calm” demeanor. But before they could say anything, Sandeep, who is not only my reporting manager but also the Project Owner and the organizer of this meeting, joined us.

The meeting started with an induction, which mostly focused on introducing me to the team. This team would be working on a fresh assignment. The others looked familiar with each other. I am sure they might have worked together recently or even multiple times. As I was the only unfamiliar face among the four people sitting in the room, I went on to introduce myself. *“Hello everyone, I am Abhishek, your friendly neighborhood DevOps. I have experience of around 1 years in the industry, I've just started my DevOps journey, looking forward to become a better DevOps engineer by the end of this project. I am a skinny man who aims to be a fat cat without having to let go of my interests. My interests are a long story for some other time.”* It was a sort of mixed feeling of accomplishment and pride to be the person who accounts for bridging the gap between development and operations. Keeping it short and precise, I handed it over to Sandeep to cruise the meeting further.

Before the sprouted seed of my curiosity could have grown a little and asked about by the young personalities sitting in the room, Sandeep took over and did it for me. The first guy is Scrum master, Sajal, confident and visionary,

who previously facilitated Scrum methodology among multiple development teams and projects, leading them to successful deliveries. Yes, this is what I concluded while listening and interpreting him, imagining myself to be a part of the next successful delivery in his ongoing streak. Sajal seemed like a guy fit for the job. I say so for two reasons: observation and experience. He talked like he knew his stuff. He was confident and funny, and his working history spoke for itself. I've known guys in the past who, to say the least, weren't exactly cut out for their job roles.

Well, for the second guy, I somehow guessed his job even before he introduced himself as the development lead. I am kidding, he was the only guy left in the room not yet introduced. His name was Adeel, and his background was impressive. Graduated from a prestigious NIT, and he had been into developing from the beginning of his career. He's the one with whom I would often be aligned in the coming weeks. With this, we knew each other, at least by our names and roles, but I wished to share a good professional bond with these intelligent guys.

After introductions took around half of the scheduled time, I noticed Sandeep quickly wrapped it up to get on with the real agenda. In his introduction, he didn't say much that didn't matter, as I had known Sandeep for quite a while, not only from the interview we had, but also from before. I had heard about his leadership and technical skills from ample people in the industry even before joining the company. He was the one I as most excited to work with.

As Sandeep connected his laptop to the projector, I took out my notebook like a school kid waiting for the action to begin. He then started his presentation, explaining the project from the technical as well as business perspectives.

"So, guys let's kick it off", he said, "I had a meeting with the client regarding a product he wants us to build: a **video consultation app** that will help patients to interact with doctors over video conferencing. This application will not only have crucial personal information but also the medical history of all the patients. Also, the system should have the capability to search for and filter the consultant doctor and schedule appointments. Now, the first thing that comes to my mind while talking about personal data and medical history is to have a secure and robust code and system. For this, we need to take some additional measures in terms of security while developing this product. Does anyone have any questions or ideas regarding the product until

now?”

“Yes, we are clear on the big picture of the product”, Sajal acknowledged. The rest of us nodded in agreement, implying that Sandeep could continue. “That’s great”, said Sandeep. He sounded a bit excited to me as he continued while looking at me and Adeel, “So, the next thing I want you both to do is collaboratively identify the technology stack we will be using to develop this product and come up with a proposal. Meanwhile, Sajal and I will continue to discuss the high-level timeline and team build-up activity. Adeel, you must also create a proposed list of available developers for this project, and Abhishek, you can collaborate with Adeel on the tech stack. You should also get acquainted with the processes we follow and come with your queries along the way.” Why wouldn’t he be excited, this was the time for action! He concluded the meeting with an ending note, “There are two things I am sure about: first, the project is not too complex, and second, we have a strict timeline of 6 months. We have to make sure that delivery is within the agreed time period.”

“Aye Aye, Sir!”, I said in my head. The excitement caused me to say it but anxiousness didn’t let it come out. Having received our tasks, Adeel and I stepped outside. Adeel asked me to meet him post-lunch as he was working on a prior task. “Meanwhile,”, he said, “you should go see Vishant. He will be working with us on this project. I’ve already informed him that you’ll be coming. He will introduce you to our processes. You both should also decide on what tech stack we should go with. Then, we’ll finalize the proposal post lunch.” As I walked to my seat, my thoughts were overriding my consciousness. I couldn’t help but think about what we were going to use: Java, Golang, React, Nodejs, Python, MySQL, Mongo, so many options to consider. I was curious about the team processes Sandeep mentioned as well. “Maybe, I’ll write it in my notebook or print it.”, I thought.

Vishant’s cubicle was right opposite mine, so finding him wasn’t a challenge. I went on to introduce myself, shared the formalities, and got on with the task. Later, we met with Adeel to draft a formal proposal.

It was just like the aura of Day One of a new project, back-to-back meetings, thoughts running, introduction with new faces, aspirations, expectations, and so on. In between all this, we all were finally prepared with our assigned responsibilities and gathered to continue where we had left off a day before.

“Welcome back team!”, Sandeep greeted, “Sajal and I analyzed the project

and the approximate number of man-hours required to timely deliver the project. Did you guys do some groundwork in terms of the technology stack we will be going forward with?"

"Yes, Sandeep", replied Adeel. "I went through the requirements in detail and totally agree with you that it isn't too complex. We prepared a high-level architecture where the approach is to divide the application into three major parts, that is, front end, back end, and database. Front end, the user-facing end of the application, can be developed on ReactJS. For the back end, user abstract logic, I opted for GoLang. And as it's a microservice approach for the back end as well, there will be two services: one will handle the employee details and the other will be a scheduler that will transfer salary to the employees' account. Finally, for the database requirements, we'll be using MySQL.", he added.

"Okay, sounds good. Just out of curiosity, why did you choose to go with GoLang for the back end?", inquired Sandeep.

Adeel smiled and said, "Well, GoLang is fast. It is compiled into machine code and doesn't require any virtual runtime. Not only this, since we are going with distributed architecture, its concurrency feature will help us a lot in scaling. Also, we have a team available who are good at GoLang programming."

"I am all good with this if Sajal is okay. I assume you guys have already discussed and planned the resource alignment. Sajal, do you want to add anything here before we start?", Sandeep asked.

"Yes, Adeel's architecture looks good to me as well. I just want to introduce the team that will be aligned with this project. Starting from the development team, we will have four developers working under Adeel, one QA, and one DevOps. Abhishek will handle all our DevOps requirements, which will involve infrastructure, build and release, monitoring and automation required from the product development to production release", Sajal said.

The following figure is an illustration of the project stakeholders:

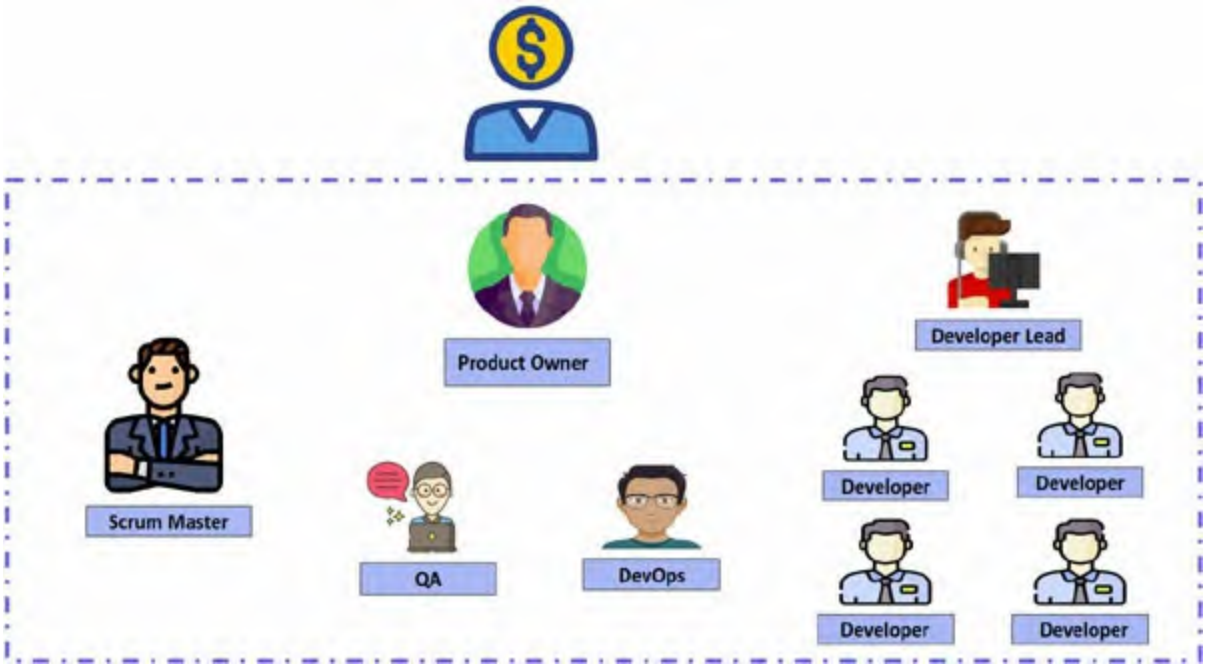

Figure 1.1: *Project Stakeholders*

With not enough to speak in the meeting and clarity around the tasks, I was mostly listening. Finally, the meeting ended with Sandeep's boosters as he roared, "Bang-bang".

Sprint-1 Retrospection

As the high-level planning meeting ended, Sajal started sharing the real ground-level planning that he outlined for the project, which was estimated to be completed in 6 months. He ballparked the development in a 12-sprint plan of 2 weeks each, and we were currently focusing on Sprint 1 to get started. On the sprint planning day, I was trying to catch up informally with the rest of the team members.

I have to admit, with the people involved in the project and planning sessions at the beginning, I thought this was going to be a smooth task. But who was I kidding? After 2 weeks, when the sprint was completed, the retrospective board looked something like this:

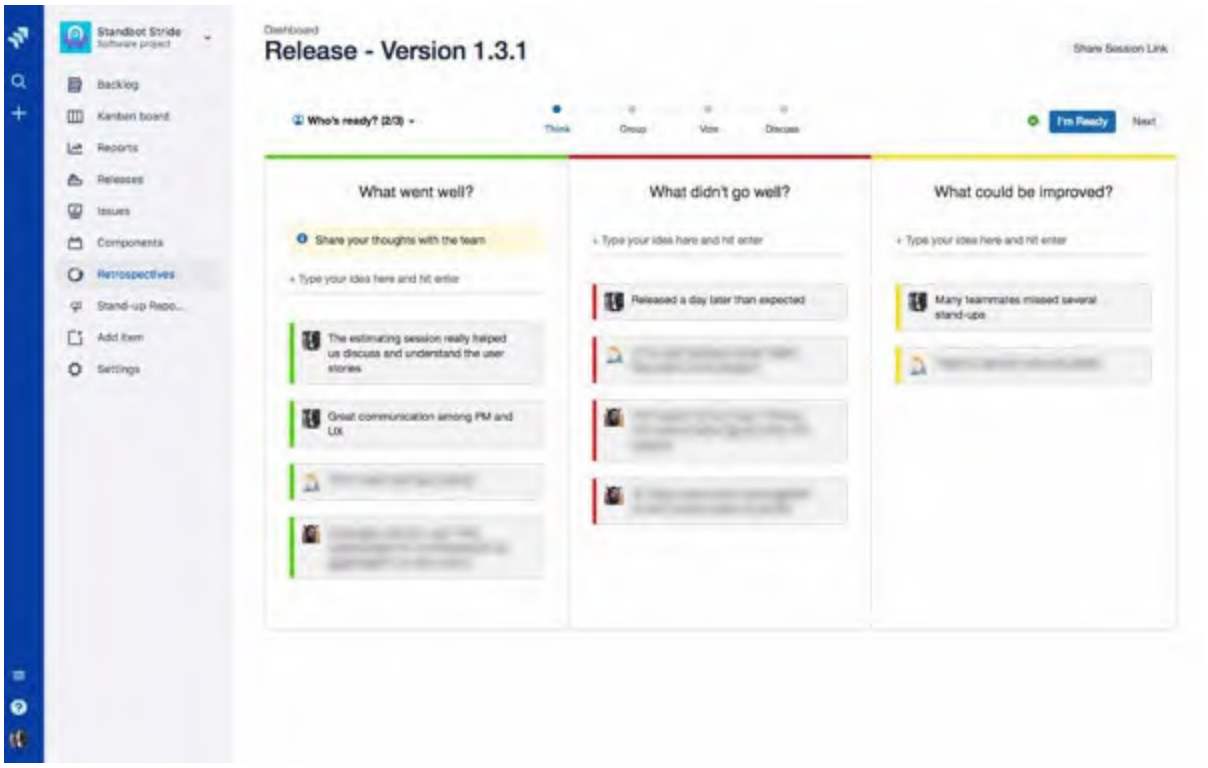

Figure 1.2: *Project Restrospective Board*

The two things that we noticed after entering the room were the whiteboard with three columns and the silence. Everyone present in the room, including me, knew the reason for the latter. Sajal finally broke the awkward silence, “With Sprint 1 coming to an end, I would like to brief the output that we delivered.”. He sounded disappointed as he further added, “We had targeted to deliver their features successfully by the end of Sprint 1, but we were able to complete only 1 feature successfully. The second feature is breaking during QA, and we haven’t started development on the third feature yet. Sandeep was also unsatisfied with the output we generated in the last sprint. So, the agenda of this meeting is to retrospect the last sprint. I want everyone to express their opinions on the same so that we can overcome the flaws and come up with better output with improved quality. Adeel, would you like to start?”

Before we go into the meeting dialogue, I’d like to add that I am the first DevOps hire of this company. Like me, Sandeep is new here as well; he joined a month before me. The last sprint was more of a noun to us than a verb as we were getting acquainted with the way things are done here. As I expected after a glance at the retrospective board, Sandeep wasn’t too happy

about it as well. In the meeting, apart from me, are Sajal; Adeel; Sonia, who is aligned as QA; and the development team, including Vishant and Harsh, both of whom I know.

Adeel: “One of the prime obstacles we are facing is increased build time because of parallel feature development. We need to create continuous features, and every compilation phase of the application takes some time; also, we need to compile code after each change, which is affecting the development cycle. This usually becomes a drag for developers. Due to this, we faced issues when a developer commits an unstable code without properly verifying, and it takes a lot of effort and time to fix the problem if they are not around.”

Sajal: “Hmm, it seems you guys are facing problems with code stability. Anyhow, let’s talk about the solution later. Let’s hear out all issues first.”

Sonia: “We’ve had to invest a lot of time addressing basic problems like insecure dependencies. If we don’t consider this check, we will have a big risk of a security breach in our application. So, we have to ensure that our code does not have any vulnerabilities.”

Adeel: “Our coding standard is not assured because there is no method and variable naming convention that makes it difficult for developers to understand code. We do have to spend a huge amount of time doing simple application checks, such as code smells and redundant code.”

Sajal: “Seems like we don’t have good-quality code. Let’s bring it into action items as well. What about testing, do we face any problems while testing code?”

Adeel: “Yes, I was about to come to this one. Well, it’s not too much, but we face problems while integrating code as well because it takes a lot of effort to find bugs and most of the time, defects are in other dependent modules.”

Harsh: “I would like to add a comment as well. We have to put a lot of work into code review to find integration problems. As a result, our development time gets affected.”

Sonia: “Conditional scenarios were also not covered properly, which resulted in a lot of bugs, and in some cases, business units were also not covered. There were many places where certain vulnerable dependencies were used, and other parts of the code were subject to potential threats. So, it would be easy for anyone to break into the system.”

Sajal: “Okay, any other issues in the testing part?”

Sonia: “Well, since you have asked, we have to execute Regression, Smoke, and Browser testing manually for each snapshot release of the code, and it takes a lot of time.”

Sajal: “Well, seems like there is a lot of scope for improvement in code testing. So, any other issues that need to be discussed before we start the discussion over these problems?”

Adeel: “I think we need to define a better branching strategy as well because right now we have only one development branch. This creates issues for one developer because another developer’s code is still in the development phase, and since they are using the same branch, developers have to wait for one another to complete their code.”

Sajal: “Well that’s a good point! Does anyone want to add anything we haven’t discussed yet?”

Everyone: “Nothing from our side.”

Sajal: “Okay, anyone wants to talk about why we’re struggling with these problems and share some ideas for addressing them?”

Light of Hope

Finally, the moment that I had dreamed about had arrived, landing like Iron Man at the Stark expo event. All the problems discussed can be handled quite efficiently using a CI/CD pipeline. By now, it was clear that these folks had not been introduced to such a system of releases. Well, now is as good a time as any to get started.

Me: “As you said earlier, we don’t get good quality and consistency in code. I completely agree with that. I also think that we have issues with unit testing, code coverage, security scanning, and even functional testing.”

Sajal: “Exactly, do you have anything on your mind to fix this, or anybody else would like to share their thoughts?”

Me: “I think we can incorporate CI into the microservice we are developing, from which the development team will receive continuous feedback that will improve the efficiency and quality of development. We can also include testing in our CI pipeline where the pipeline itself can conduct basic testing, which would allow the QA team to concentrate on more pressing business

requirement testing. I have a good understanding of CI and can guide you."

Sajal: "Sounds good. Does anyone have any questions or views on this strategy?"

Adeel: "I like the idea of incorporating CI. It will surely increase our development and testing efficiency. I think this is a good time to pick this up. I have been reading and found that it increases the SDLC process efficiency by a huge amount. I had this discussion with Harsh some time ago. We were only waiting for somebody with expertise in it."

Sonia: "The plan sounds good to me."

Sajal: "Great! So, Abhishek, do you have any plan in your mind as to how we can implement this process?"

Me: "Yes, I do have some points in my mind, but I would like some time to evaluate them further so that we can implement them with the best practices."

Sajal: "Sure, so you can draw a detailed plan, and we can discuss it in the next meeting. I will share the next meeting invite."

"Great, I will try to come up with a plan of action.", I affirmed as my team saw a light of hope.

And here, the real challenge started for me as I had to draft a proposal. Thankfully, I had already been involved in the sprint, which made it easy. Sometimes I feel so grateful to our open-source community, especially the pioneers like Eric Raymond, Linus Torvalds, and Martin Fowler who laid the foundation. Where would we have been without them? Our whole career exists because of the decisions we made and the actions we took. The DevOps movement is just another fruit of their humongous tree.

We are planning to include bits of trivia at different places in the book. The following links are part of that.

CI Trivia

- **https://dzone.com/articles/continuous-integration-and-its-whereabouts**
- **https://en.wikipedia.org/wiki/Continuous_integration**

It was worth spending the rest of the day, till late at night, to draft a proposal to streamline the application build process by introducing CI. In fact, I

wanted to take it one step ahead by including best practice suggestions. Now it was time to showcase my proposal in the next spring planning, which I was sure the team was ambitiously waiting for. As expected, Sajal asked for the same as the first thing in the Spring-2 planning meeting.

Me: “Yeah, I’ve put together a few points and strategies for that. Since we have all the components in ReactJS and Golang, we need to build two forms of CI pipeline, one each for applications in ReactJS and Golang. The approach and tests will be the same, but the tools to build and test would differ depending on the type of application.”

Adeel: “Okay, but what CI checks are we going to inculcate?”

Me: “We will include the standard CI checks for the application, like code stability, code quality, code coverage, unit testing, security testing, and functional testing. We will also use SonarQube for software review.”

Adeel: “This sounds pretty good. Would you help the team as well in gathering up all the thoughts we are getting about the CI checks? Also, it would be great if we dedicate 1 hour every day to the current sprint. It seems as if we are going to carry out almost all the checks that we mentioned in the previous meeting.”

Me: “Yes, except SonarQube, which is a new addition to the list. Well, among the multiple CI automation tools available, I have chosen Jenkins. I know we all are curious to know about this, but I will be discussing it as I go ahead with the phases of exploration, planning and implementing them one by one.”

Sajal: “Great. This seems interesting, but how are we going to implement this in all applications? Also, any specific reason for choosing Jenkins?”

Me: “It is a common automation tool that is not only limited to CI/CD but can also be used for other tasks. It has a plugin-based architecture that will help us implement CI for various types of applications. In other words, the CI/CD tool is like a butler, and we are his employer who can ask him to work on any task. Also, I have prepared a comparison between the available options to choose the best one.”

The following image shows the popularity of various automation tools:

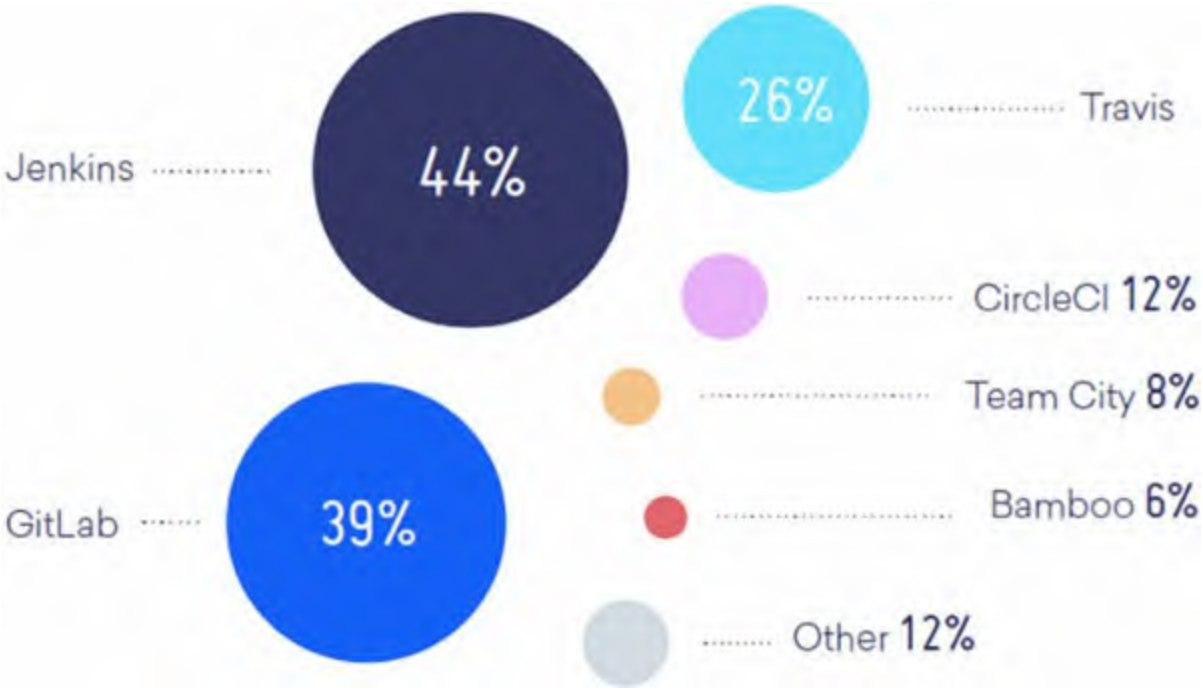

Figure 1.3: *CI Tools Market Share*

Sajal: "Great, I don't have further questions. Does anyone have any other questions?"

Adeel: "No, the plan looks good to me."

Sajal: "Okay. Adeel, do you have something in mind for branching strategy since this is also one of the blockers we are facing?"

Adeel: "Yes, so I was thinking about this. There are a few branching strategies I am familiar with and had to choose one that would fit here. This is what I've come up with. We can have a common development branch from which developers can create their own feature branches. As soon as they are done with the features, they can raise a PR(Pull Request) for the main development branch. Since we will have CI in place, no buggy code will be merged in the development branch. After the testing of features, we can raise the PR for the production branch from the development branch."

Me: "Great, I think this is a good approach."

Sajal: "Awesome! Now we have a strategy for branching and CI, so let's implement it. Then we will evaluate the progress in our next retrospective meeting. This sprint, we will be fighting our failures. And after seeing your passion, I am confident that the current sprint will be a success."

While riding back home, I was overwhelmed. There was so much to do. I was

restless, nervous, curious, and at the same time, puzzled. In fact, I'd go as far as to say that I was excited. You see, this is no joke. To get an opportunity to design the CI/CD pipeline from scratch for a project, on one's own is a rare opportunity. I was surely going to make complete use of it. I kept thinking about where to begin. Then I thought, let's keep it simple and not overthink. This is what my mentor taught me. I know what we are trying to accomplish, I know the problems we are facing. Let's break it down.

We'll divide the whole pipeline into three steps: pre-deployment integration checks, intermediate actions, and post-deployment integration checks. In the first part, we'll cover basic code sanity, like code quality, code stability, and code coverage. In the second part, we'll take care of the artifact. It will focus on actions surrounding the actual build, that is maintaining dependencies, publishing the artifacts, handling versions, implementing rollback strategy, and so on. The third part will include integration tests like smoke and regression tests. Notifications and alerts will be configured at each step to give an overview of the pipeline. Wow! That eased my burden. Having thought this out this much during a metro journey, I was pretty much content. Now, the actual planning was to begin.

Conclusion

In this chapter, we discussed the general problem people face while starting a new software development project, like sprint planning, application development, application testing, and release planning. We also looked at how CI helps us to solve such problems in an organized and efficient way.

The next chapter will explore the concept of CI. We will break it down into multiple steps and discuss them individually, making the idea clearer.

Questions

1. What are the different challenges while starting a software development project?
2. What are the different types of CI tools available in the market, and on what factors do we choose a CI tool?
3. What is continuous integration, and how does it help us?
4. What is sprint and sprint planning?

CHAPTER 2

Continuous Integration

Introduction

“I need some sticky notes, a marker, and a whiteboard with magnetic pins,” I thought as a big gulp of espresso slid down my throat, “I’m going to nail this CSI style.” This is the morning I have to start working on the CI plan. The gears in my brain were already turning. “Pre-deployment checks, huh.” Well, I do remember what Adeel said. If I have to break it down, I’d say there were two major problems: code compilation was taking a huge amount of time as they were doing it manually after each feature change, and bug fixing was a painful process as multiple developers were involved. Also, there was no automated check-in place for recurring, annoying bugs. Since I am early today, I should scurry to my bay and get a consolidated plan in place for pre-deployment checks. Also, I’ll send a calendar invite to Adeel for a discussion later in the day.

Structure

In this chapter, we will discuss the following topics:

- Pre-Deployment checks
    - Code Stability
    - Code Quality
    - Unit Testing, Code Coverage, Security Testing
- Intermediate Operations
    - Artifact Management in Continuous Integration.
    - DB versioning while making database-related changes
- Post Deployment checks
    - Smoke Testing

    - Regression Testing
    - API Testing
- Notifications
- Branching Strategy

Objectives

After studying this chapter, you should be able to understand the concept and importance of code quality, code stability, unit testing, code coverage, and artifact versioning as well as management using Continuous Integration. You should also be able to test strategies and their implementation. You should be able to know the importance of branching strategy and how to implement it.

Set up

“What is the most important aspect of testing?” I asked a **Quality Assurance** (**QA**) once, who was my colleague and friend. He said, “What is the most important aspect of eating? To make optimal use of food, provide necessary backing to human functioning and flush away waste. It applies to everything.” I know it is corny, but there is some truth to it. Now, automated testing can’t replace QA (yet), but it can make both the QA’s and developer’s jobs easier. In my checks/tests, I’ll cover these problems: code stability, code quality, Unit testing, code coverage, and basic security testing. If I had to explain these checks, it’d go something like this:

Code stability testing is different from stability testing, which aims at testing end-product endurance over a period of time. Code stability protects the codebase from rogue changes that might hurt more than they will help. It aims to keep the build job stable.

Similarly, **code quality** check involves keeping the code standard high. We need to make sure that Linting is proper, documentation is in place, complexity is as per standard, reusability is easy to follow, known bugs are identified and handled, and so on.

Unit testing, as the name suggests, focuses on individual features to make sure they are stable. Making this a part of CI is essential in ensuring a reliable codebase.

The idea behind **code coverage** is simple; this metric shows how many lines of code are covered by the test cases. Having this strengthens one's confidence in code quality. It is a sort of double-checking that raises the standard of code.

Basic **security testing** will ensure that code is not susceptible to commonly known threats. This test involves checking things like ensuring that no key is committed in the codebase, code cannot be exploited with query manipulation, and that the exceptions are handled as per security policies.

I was convinced that these checks would not only cover Adeel's issues but Sonia's as well. I thought I should sit and draft this into a proposal. Adeel had also accepted the invite. The meeting was on.

The following figure illustrates the common checks in CI:

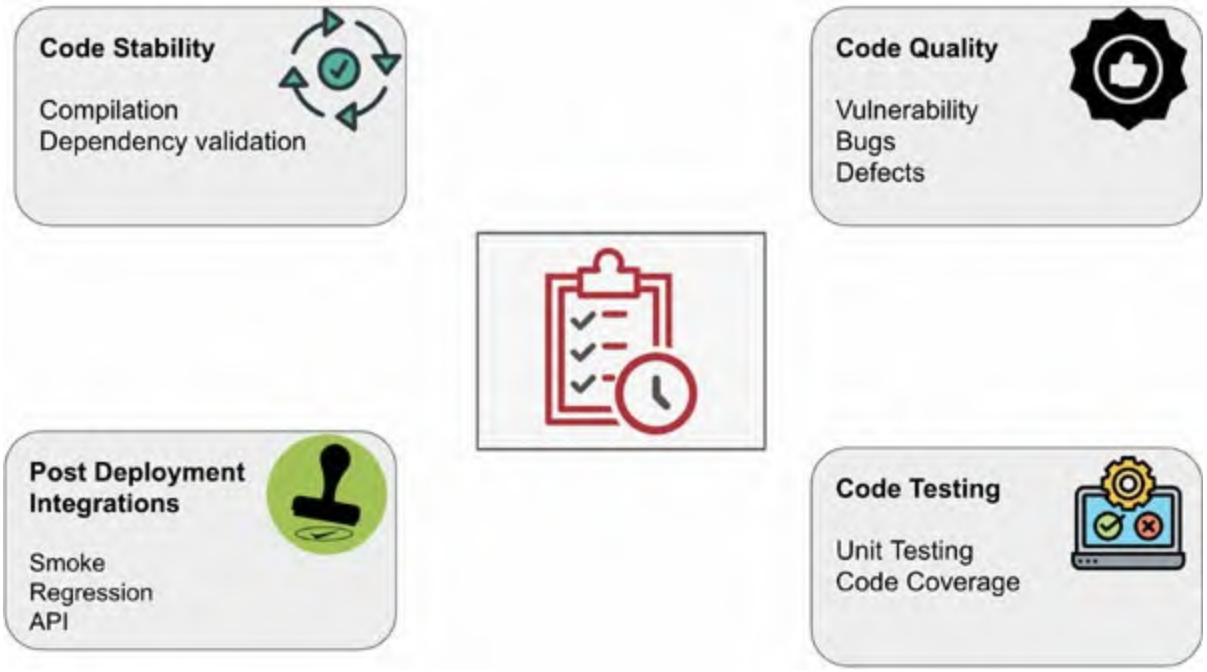

Figure 2.1: *Common checks in CI*

About 10 minutes before the meeting, Adeel pinged me to come to a conference room where he had just finished a meeting. I had my proposal ready and thought that Sonia should be a part of this meeting as well. Though it was short notice, thankfully, she joined us.

Pre-deployment Checks

Me: "Hey Adeel, how are you?"

Adeel: “I am good, how about you?”

Me: “I am doing good. Just wanted to discuss something about the CI pipeline we are going to implement. Just to inform you, Sonia will also be joining us. I think it is imperative that she does.”

Adeel: “Alright, please have a seat. I’ll step outside for a minute to pick up this call; will be right back.”

Me: “Sure”

Adeel left the room.

While waiting for them, let me quickly verify that the order is correct. Today, we will be discussing pre-deployment checks. It includes code stability, code quality, unit testing, code coverage, and security testing. It should be enough for today.

Adeel and Sonia enter the room.

Me: “Hey guys!”

Sonia: “Good afternoon to both of you. Abhishek, you seem to have some interesting things to discuss. Let us begin as I may have another meeting later.”

Code stability

Me: “Agreed. So, as I was saying before, this is about the CI pipeline I am working on. I will start with code stability checks because it is affecting the majority of development productivity. Adeel also pointed it out specifically during the last scrum meeting.”

Adeel: “Yes, that is true; we are facing lots of challenges due to uncontrolled changes. So, whenever a developer commits unstable code, the job of the whole development team gets affected. Sometimes it takes a lot of time to figure out the issue that could have been avoided with a proper workflow.”

Me: “Yes, I understand your pain. That is why I wanted to start with code stability. So, I have some ideas on which I want your suggestions as well.”

Adeel: “I will be happy to help you. But before that, I would like to ask what exactly is it we are going to do in code stability.”

Me: “Since the stability of code check means checking whether code is in a compliable state or if new changes are breaking already compliable code in

any way. So, in code stability, we will compile the application code to check whether it is breaking the application. Does that make sense?”

Adeel: “It does, so how can I help you with this?”

Me: “I would like to know what the frequency of code stability checks should be, i.e., when exactly we should check code stability.”

Adeel: “I think it could be nightly or in the evening, so the developer can check the stability in the evening before leaving for home while changes are fresh in his memory. It would save a lot of time as well.”

Me: “Hmm, I have a different proposal though; hear me out. According to me, it should be on every commit so the developer can fix the previous code before starting to work on the new feature. Also, if the developer will check the stability report in the evening, they may have to make a lot of changes in the code that was developed during the entire day. It will hold them back late in the office. We should avoid that. Instead, we can follow the golden principle of continuous integration, which is continuous feedback. Any thoughts?”

Adeel: “I haven’t thought in this direction, but I must say I agree with you completely.”

Me: “Great! So we have decided on the frequency. Now, the other question is whether it should be a pull-based or push-based mechanism.”

Adeel: “Sorry, you will need to explain that.”

Me: “Sure. You might have read that there are two types of mechanisms, that is, pull-based and push-based. In the pull-based mechanism, our CI server will continuously poll the git repository of the application to see if there are any new changes. But in the push-based mechanism, we will configure a webhook in our git repository that will send an event to our CI server for every commit or action in the repository.”

Adeel: “By listening to the solutions, I think the push-based mechanism will be better. In the pull-based method, there would be unnecessary load on our CI server for continuously polling the repository at certain times. Moreover, the change is happening at the repository end, so it would only be logical for the repository to trigger an event instead of the other way. What do you think, Sonia?”

Sonia: “I agree. But is it always possible to configure a webhook? I have

heard it could be tricky when there are firewalls in place."

Me: "Well, Sonia. There are ways around it. It is all in the plan. Good point though. Before moving forward, let us discuss notifications. This must be configured at each level, so we might as well decide how to approach it now. In case a check fails, whom should we notify?

Adeel: "I think we should notify the developer who is committing the code."

Me: "Yes, you are right, but you should also get notified as you are the development lead. So, in case of such an event, you know whom to contact."

Adeel: "Right, you can add me in the notification as well, no problem."

Me: "One last thing. It's not a question or suggestion. I just wanted to share one idea with you. I am thinking about making a leaderboard for code stability in which people will have a ranking as per their stable commits. I think this would motivate developers to test their code thoroughly before committing it. Do you think that it is a good idea?"

Adeel: "Yeah, it's a great idea. It will encourage healthy competition between developers. I would like to know how you are planning to make it, but let us do that in a separate meeting."

Me: "Sure, I have something in mind; we will discuss it at length. For now, let us move on to code quality checks."

Note: In today's software-driven climate, the best tech companies — Facebook, Amazon, Netflix, and Google — are releasing software updates thousands of times a day. Take Amazon, for example. In 2013, the company was doing a production deployment every 11.6 seconds. By 2015, this had jumped to 7, so there is no telling what cadence could be reached in 2021.

Code Quality

Adeel: "For code quality, I have a few points too. You should continue with your proposal; I will speak after."

Me: "Okay. Most of our requirements from here on will be covered by SonarQube. It's just a tool at first look, but its features make it worth a lot more."

Adeel: "I have heard of it. Could you explain how it fits our requirements specifically?"

Me: “Well, let us take an example of Java. One can find many static code analysis tools to put checks on all aspects of programming, like checksum, find bugs, PMD, and so on. They all give their focus areas with separate reports, which are continuous to say the most. The thing is, it is neither detailed nor regulated. By detailed, I don’t mean the reports aren’t detailed enough, as they are. Allow me to explain both detailed and regulated. SonarQube is installed on a server and has its database as well. Every time a developer commits or wants to analyze the code, they can just do it, and the code passes through a multitude of tests like duplicate code, potential bugs, architecture, and design checks, code complexity, and code smells. All these different tools can be added as plugins to SonarQube. These test results are stored in the database and are displayed with proper visualization on the dashboard. You can see the changes in code, track bugs, view activity over time, and do much more. So, we not only have tests but investigation tools as well at our disposal. Let me remind you that Java is just one language. SonarQube does this for a lot of them, including Golang and ReactJS. Continuous inspection cannot get better than this. Additionally, we have regulations. There is a quality gate, which is a set of standards a code must adhere to for it to be approved for production. It can be configured based on projects. Multiple projects can run at a time.”

Adeel: “It is impressive, no doubt. Just one question: Is SonarQube the only choice we have?”

Me: “Well, no alternative provides all these features, that too for free, yet. We have got an amalgamation of various test plugins, activity tracking, and regulation imposition as well as an overview of all of this.”

Sonia: “This will help me a lot as well.”

Me: “Yes, which is why I invited you; please share your views”

Sonia: “As I said in the scrum as well, a lot of my time goes into finding issues that were caused by basic coding errors like unhandled conditionals and insecure dependencies. I am assuming SonarQube can take care of those as well. This will allow us to focus on other important areas and save a ton of our time. I sure do wish now that it is as you say it is.”

Adeel: “Yeah, me too. The points that I was trying to put earlier are already covered now. One question: how many code quality checks can we expect in the case of Golang and ReactJS?”

Me: “SonarQube goes deep. There are several rules for bug detection, vulnerability detection, and code smells, which refers to smelling something fishy based on certain factors. And of course, there is linting for both Golang and ReactJS. Code smells, though, can help lead to deeper problems that might end up wasting a lot of time at a later stage. For example, it is a common occurrence for a developer who has worked on a different language in the past to just start using libraries and modules for the tasks that can be done without external help in Golang. They will end up writing more complex code than required. SonarQube can help in such cases as well.”

Adeel: “This looks good, Abhishek. I must say. I am excited to see this in implementation.”

Me: “We still have a few things to discuss.”

Sonia: “Yeah, sure. How about we keep the rest of the discussion for tomorrow? I have got another meeting in 10 minutes.”

Adeel: “Well, in that case, I think we have to conclude. What else is left on your agenda, Abhishek?”

Me: “Well, there is unit testing, code coverage, and a little bit about development-side security testing.”

Sonia: “Well, I would like to be a part of these as well. I will be here tomorrow.”

Me: “I will include you in the invite.”

Adeel: “Alright. Thanks guys, see you tomorrow.”

“Well, it went exactly as I expected. How often does that happen?,” I thought. “In all its entirety, we are bringing DevOps to a no-DevOps project. What else could I have expected?” As I strolled down the hallway, I saw Vishant sipping that afternoon cappuccino by the coffee machine. “Hey, buddy! What’s going on?,” I inquired. “You see that gardener watering the plants, Abhishek?”, he asked. “I do, yes,” I replied, intrigued. He continued, “Such a simple job, isn’t it? Just water them once or twice daily, and they grow all by themselves with little to no effort.” I thought that was the understatement of the year. Gardening is not an easy job at all. We have to consider the location, soil quality, nutrients, plant type, and so on. Even watering is done through different techniques. It requires a lot of patience and care to grow a plant. But I wanted to hear what he would say next, so I nodded and let him continue, “Imagine if he was asked to go through all the

plants that he planted in this garden and kill all the bugs individually that are living in each of them, eating them from inside. What would he do?" "I think they have pesticides for that," I grinned. Vishant turned to me and paused for a second, which felt like minutes, and then we both broke into laughter. "I have been fixing bugs all day." he finally got to the point. "I understand your frustration. It's part of the job, what can we do, huh," I consoled. "Hey, how is that CI/CD plan coming along? I have great hopes for that, man," he exclaimed. "Well, that might help the whole team a lot in bug hunting. I was discussing it with Adeel in the meeting just a few minutes ago. Once it is implemented, we will be able to tell more." I explained. He bumped my shoulder and said, "Cannot wait man. Looking forward to it."

People were waiting, expectations were building. Everything was exactly where I wanted. My past experience had given me some leeway. The thing was that once SonarQube is set up, most of the issues with pre-deployment checks would disappear. I knew this, and that people would find it quite convenient. Please refer to the following figure:

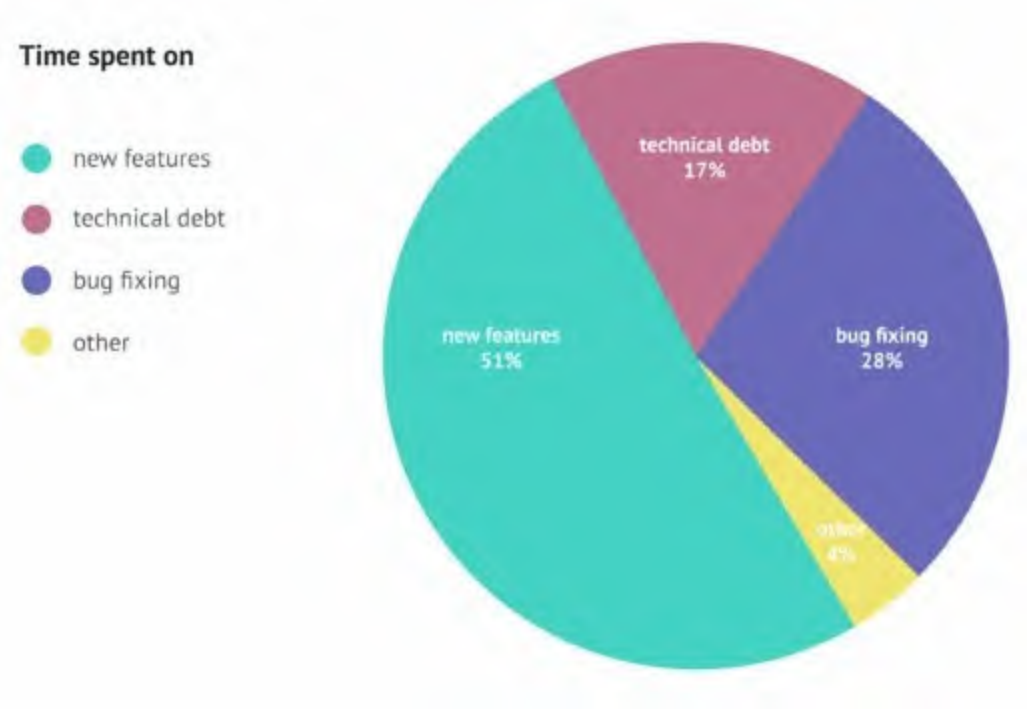

Figure 2.2: *Pie Chart: Developer time distribution*

Note: Developers spend 45% of their time fixing bugs or addressing technical debt vs building new features.

Testing, Code Coverage, and Security Testing

Next morning in the meeting room:

Me: “Hey everyone.”

Sonia and Adeel: “Hey Abhishek.”

Me: “Okay, let us keep this meeting brief. For unit testing, our developers will have to write the test cases, yes?”

Adeel: “Yes.”

Me: “Well, in that case, all we need to do is provide the details of the source code and tests in a SonarQube properties file and let it run. We might need to run some tests separately and provide Sonarscanner with the report to publish in some cases as well. Either way, the reports will be available to view on a dashboard, and we will be able to track changes as well as the progress that our code base makes over time. Isn’t that something?”

Sonia: “There are a few terms I am not familiar with per se. Would you mind explaining? Let us begin with code coverage. What exactly does it mean?”

Me: “Hmm, you know better than me that the whole purpose of testing is to reduce unhandled exceptions, right, unintended behavior of our code, if you will. No matter how good our tests are, they won’t be able to analyze the areas of the code that are not covered in the test cases. Code coverage precisely shows that: how much of the code is covered in the test cases. This way, we’d know what areas are in the dark and can write test cases for them. With time, we can also see these reports being visualized in the SonarQube dashboards with time.”

Sonia: “Adeel, don’t we already have a lot of tests in our fleet already. It should cover most of the codebase. Why do we still get so many bugs?”

Adeel: “That is a good question. Sometimes there is a method use case or a conditional branch that is left out of the tests. Usually, those are the bugs we find the hardest to tackle.”

Me: “I might know why that is. Adeel is right, but not completely. It is not as rare as one might think, which is why there are multiple code coverage methods. It can quite often be a possibility that line coverage for a particular feature is 90% but condition coverage is just 50% as it fails half of the time. We must take these things into account before moving forward.”

Adeel: “That makes a lot of sense.”

Sonia: “Agreed. My next question is about Sonarscanner. What is that? Is it different from SonarQube?”

Me: “Oh yes, it is a client-based application. We can configure it to run project analysis by putting appropriate values in the configuration file. Once complete, it sends a report to the SonarQube server for processing. It will become a lot clearer once I implement it and give a demo.”

Adeel: “So, we need Sonarscaner every time we have to run tests?”

Me: “Well, Sonarscanner is one of the scanners used for project analysis. There are others as well, like maven for java, that you may use as per your need. But the SonarQube server does require a scanner for project analysis.”

Adeel: “And all of this is dictated and managed by the SonarQube server?”

Me: “Yes, that is correct.”

Adeel: “Okay, so that is two down, one to go.”

Me: “Now, let’s move on to security testing. To be honest, there is not much to talk about here either. SonarQube can take care of this as well. The plugins for respective languages do come with vulnerability rules, which are created based on threats and vulnerabilities defined by standards like CVE and OWASP. Of course, there are others as well, but an important thing to note is that if there is vulnerable practice being followed in code, it will appear in the analysis report.”

Adeel: “Could you, maybe, give an example?”

Me: “Suppose a developer has mistakenly committed a password in the code or an access key. It can also detect areas in the code that may be prone to attacks, like SQL injection or cross-site scripting. We must also encourage our developers to follow secure coding practices but, to be honest, that is another topic.”

Adeel: “I do not have any more questions. I must say Abhishek, all this looks pretty good. Let us implement it as efficiently as it sounds.”

Sonia: “There’s the manager we all know Adeel to be!”

Please refer to the following figure:

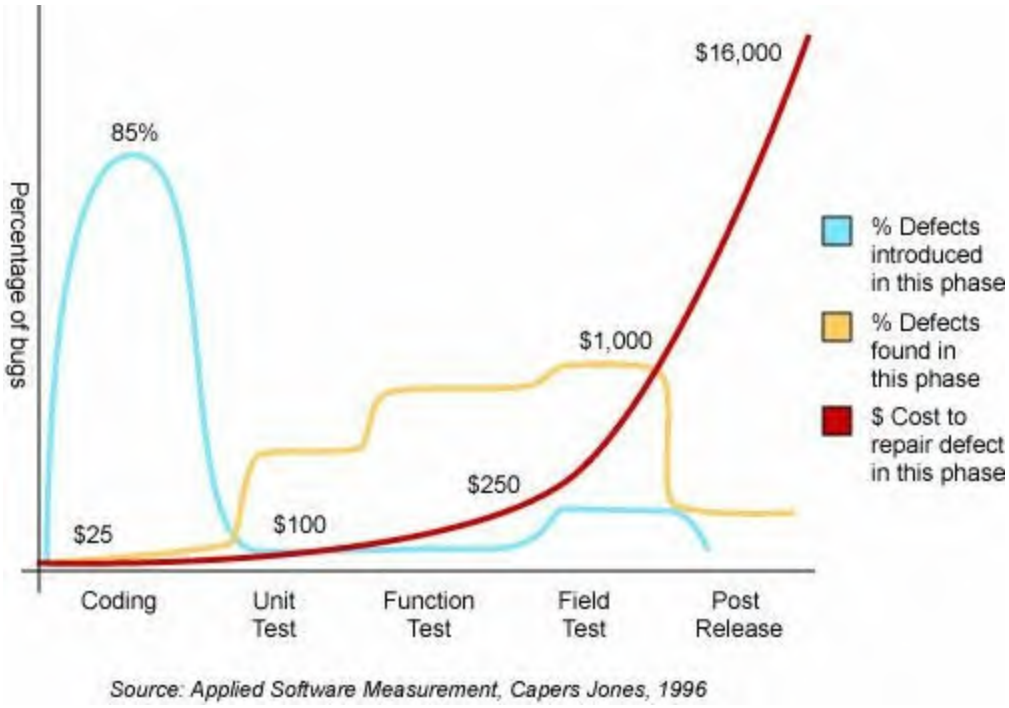

Figure 2.3: *An interesting or rather compelling comparison*

Fact: In 2016, software failures cost the worldwide economy $1.1 trillion. These failures were found at 363 companies, affected 4.4 billion customers, and caused more than 315 years of time lost. Most of these incidents were avoidable, but the software was simply pushed to production without proper tests.

Intermediate operations

With all the thoughts running through my head regarding the CI checks and believing it to be a positive omen that I was getting the answers to all the issues that were discussed in the scrum meeting, I was feeling relaxed, but my mind got hijacked by the thought of managing the growing artifacts. I was thinking of the challenges that were about to arrive with frequent builds and the management of dependencies. Lost in my thoughts, I did not realize that Adeel was standing right beside me.

Artifact management

Adeel: “You look puzzled, is there something important going on in your head?”

Me: “Well, yes, there is. What is the timing? Just the guy I wanted to talk to. Post CI implementation, we will be having more frequent continuous builds running around; I am thinking about managing the dependencies.”

Adeel: “If I am right, you are talking about dependency libraries and our application artifacts.”

Me: “Yes, I am talking about both and in fact more. I have created a list of queries around it. Let me share it with you.

It will require a good amount of bandwidth, data to be transferred and time to trigger builds repeatedly. This will increase the overall cost of triggering the builds as the dependencies will be downloaded from remote repositories.

Duplicate copies of dependencies will be common across different environments.

The stability and reliability of remote systems from where we are downloading our dependencies also need consideration. And most importantly, what if we need to roll back to the previous version? Do we need to roll back the changes in our code and trigger the build again?”

Adeel: “Hmm, looks like you have some genuine concerns about artifact management. I would definitely recommend that you look more into this area. If we manage the dependencies and artifacts right, we will surely be able to build a redundant build process that would be faster too.”

As it was gratifying to listen to Adeel, it also meant that I needed to do some research. With a mutual agreement, I planned to spend time exploring possible options that are specifically available to solve the preceding requirements and may provide more features that were still unknown to us.

After spending a good amount of time in exploration, including reading blogs and holding discussions, I figured that multiple options are available to manage application artifacts and dependencies. Among those options are JFrog Artifactory, Sonatype Nexus, Pulp, and Archiva Cloudsmith Package. It was just like what they say, “The more options we have, the more difficult it is for us to choose.” Now, to deal with this problem, I followed the golden approach: sorting and elimination. I narrowed down my primary requirement to a tool that is primarily open source and supports most of the programming language to be future-ready.

It took a good couple of hours to narrow down my search and arrive at the final two most feasible tools based on the factors like age of the tool,

stability, technology support, requirement fulfillment, and of course, the community followers and support. Refer to the following figure:

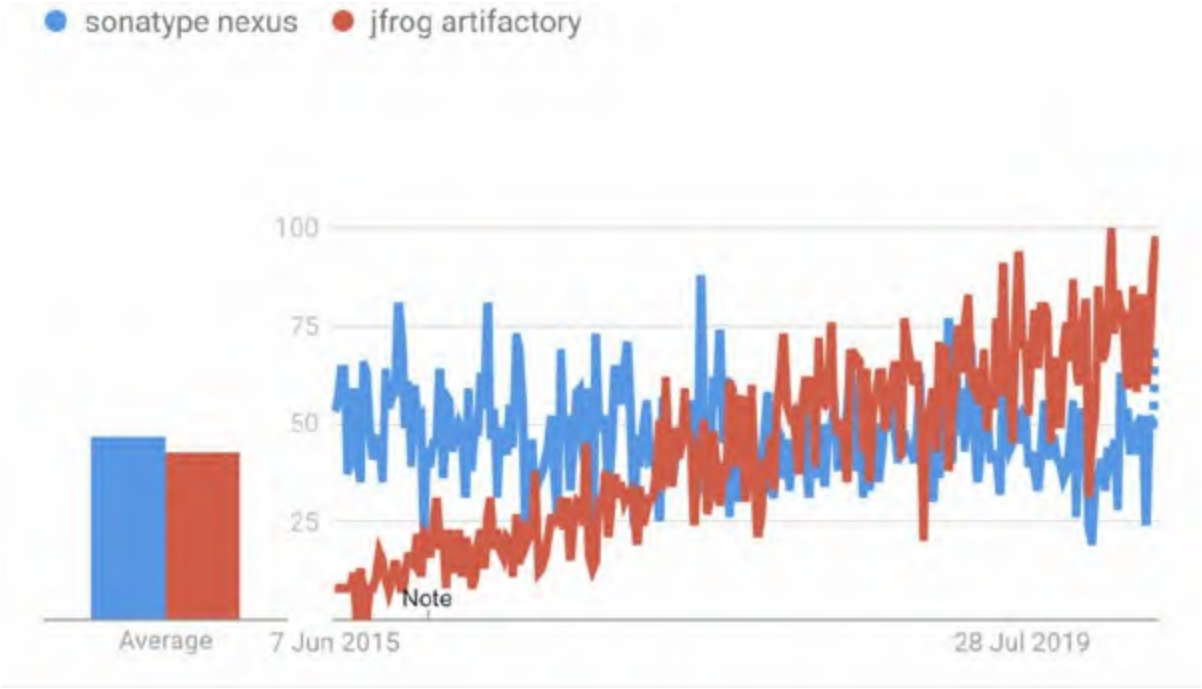

Figure 2.4: *A chart comparing nexus and Jfrog adoption over time*

With all my initial research and findings, I was ready to present both the solutions in the next scrum discussion, but before that, I discussed it with Adeel to get his views and ensure that it can serve our purpose.

“Yes, this will serve our purpose; in fact, I have heard about these tools from my previous colleagues, and I am sure we can pick and use any one of these as our artifacts management solution.”, Adeel replied satisfactorily, which boosted my morale of proposing the solution.

In short, the artifact solution will be capable of handling the following requirements:

- Reduce network traffic and optimize builds
- Reliable and consistent access to remote artifacts
- Integration with build ecosystem
- Security and access control
- Ability to scale and handle failover
- Open source licensed

DB Versioning

I can still recall the moment when I was attending a seminar, the speaker used a phrase, “You just can’t restrict yourself to what you are exactly searching for.” And yes, it was true, this was happening to me. While exploring artifact management, I stumbled upon a query, “How to handle Database Updates”. Although I was searching for the former, the latter was also revolving in my mind. I decided to book my next day searching how updates are handled at the data level, be it **Data Definition Language** (**DDL**) or **Data Manipulation Language** (**DML**), and what if we need to rollback any database change along with the application rollback, or what if any database update fails.

Here again, on the same desk with the same spirit as yesterday, I continued where I left off. One thing I was very clear about was that keeping track of database updates for an application is not an easy task. It may tend to have differences among multiple environments. And the situation may become worse when it is caught in the production environment.

I called up a meeting with the development team to discuss it further and introduced the team to the concept of Database as Code. Upon further discussion with the team, I explained how we will treat database changes just like application code. This will require scripting of every change required in the database and have its version controlled along with the application release.” With the team nodding, I further added, “It will not only help us update the database in a managed way, but it will also version the changes and state of changes applied. So, in case we need to roll back to the previous changes, we can undo the changes to the last stable state.”

I listed down other merits of following this approach:

- No problems with the database schema mismatch in multiple environments
- Increased visibility to database updates
- Possible to set up new database instances from the scripts

I explored how the structure of database scripts is generally maintained. An example can be seen in the following figure:

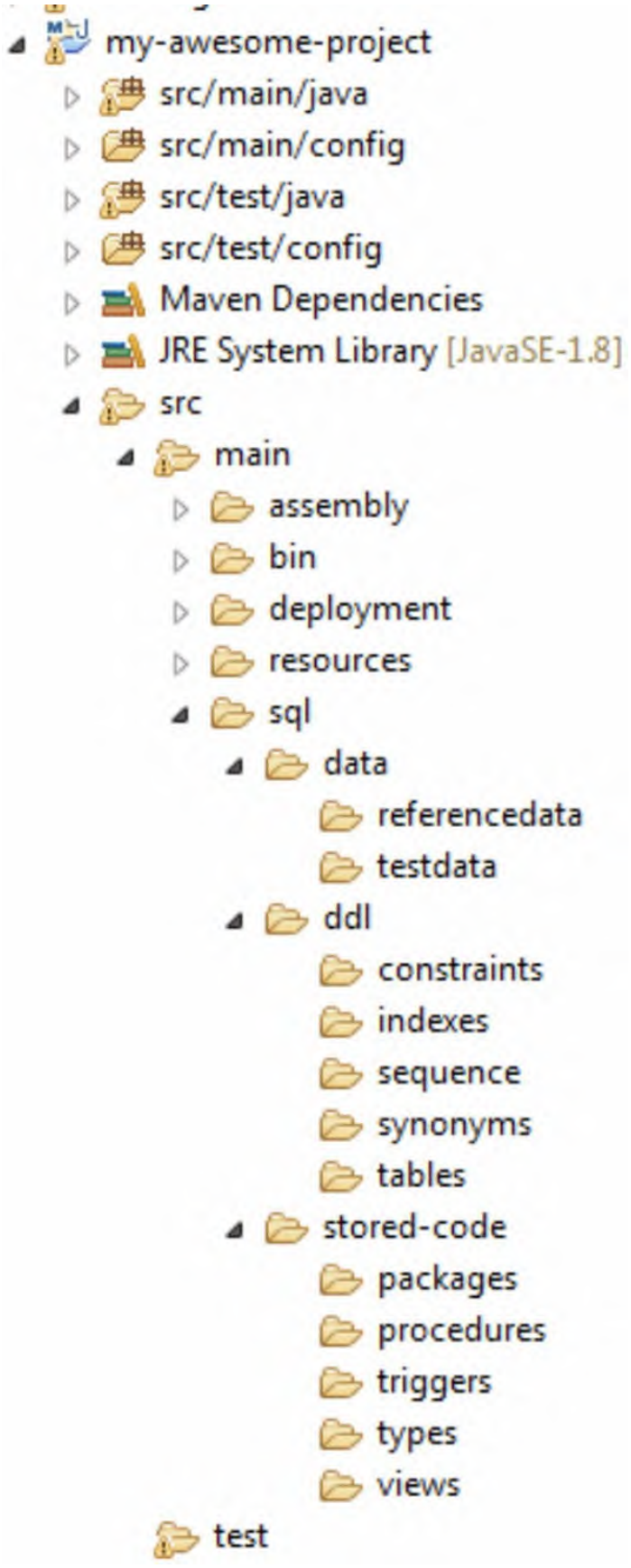
my-awesome-project
src/main/java
src/main/config
src/test/java
src/test/config
Maven Dependencies
JRE System Library [JavaSE-1.8]
src
main
assembly
bin
deployment
resources
sql
data
referencedata
testdata
ddl
constraints
indexes
sequence
synonyms
tables
stored-code
packages
procedures
triggers
types
views
test

Figure 2.5: *An example structure of a project*

There are few players in the industry to manage DB updates with Flyway and Liquibase, both going neck to neck in open-source distribution. The adoption chart for Flyway and Liquibase is shown in figure 2.6:

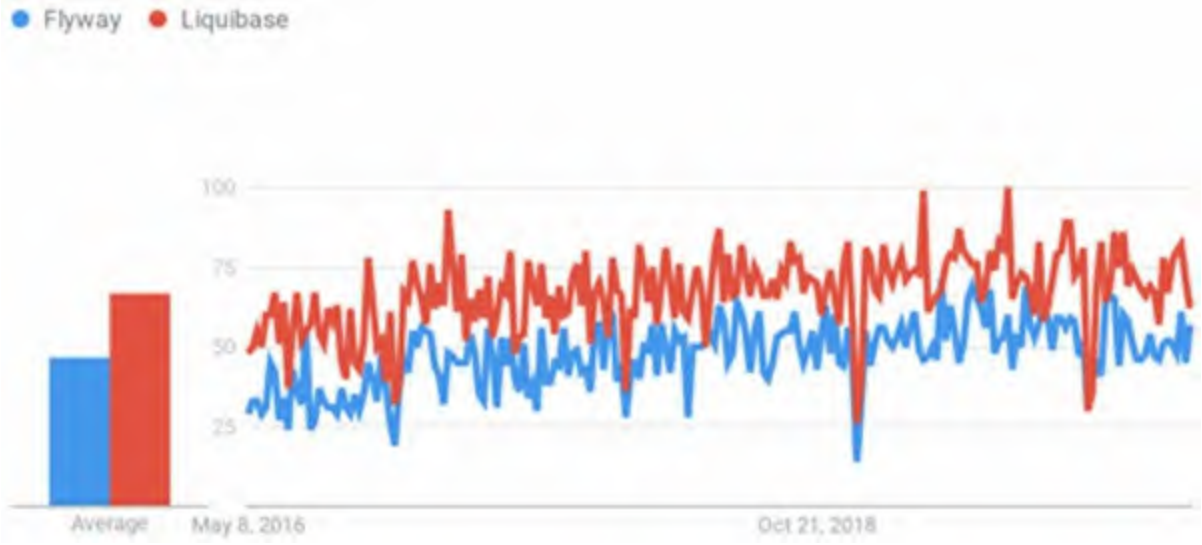

Figure 2.6: *A chart comparing Flyway and Liquibase adoption over time*

Fact: In the 2020 State of Database DevOps Report, 64% of respondents said the speed of database changes, the developer's bandwidth, and reducing the risk of losing data during deployments were the top drivers for automating the delivery of database changes.

Post-deployment integrations

I was quite relaxed when the CI implementation plan was almost finished. So, I ran through the checklist and started to check the boxes for the tasks that were covered. Suddenly, I realized that we haven't covered the post-deployment checks. I had kept it aside earlier during the planning stage as I needed some advice from QA regarding it. Little did I know that it would completely skip my mind.

Smoke testing

The next day, I went to meet Sonia after reaching the office.

Me: "Hey Sonia, how are you doing?"

Sonia: “All well, how about you?”

Me: “I am good as well, thanks for asking. I came to talk to you about the post-deployment checks in our CI pipeline. Since our pre-deployment CI checks and intermediate operations like the artifact versioning and DB updates have been finalized, the next big thing is post-deployment checks. I am thinking of including smoke testing, regression testing, and API testing. I would like to discuss each part in detail with you. Let us start with smoke testing. Let me start with my understanding of smoke testing, and you can correct me if I am missing something.”

Sonia: “Sure, go ahead.”

Me: “The term smoke testing originated from hardware development, where if you power on a circuit board for the first time and if you see any smoke rising, you’ll know it’s broken.

Sonia: “Right. The same philosophy applies for software development as well.”

Me: “Yes, I agree. In software development, if the application comes up and does not die immediately, then it means that the application passed the smoke testing.”

Sonia: “I am afraid you are not completely right. Let me explain it to you with a simple example. Whenever there is a pipe fitting, there are multiple tests done just to verify that the pipe is not leaking anywhere. If we correlate this with software development, when an application gets started, multiple user flows are executed on it to validate its basic functionality. So, smoke testing includes many benefits, like testing of basic application functionality and ease of deciding whether further testing is needed.”

Me: “Great, this is something interesting. Would you please explain what we mean when we say basic functionality?”

Sonia: “That is a very good question. The basic functionality varies from organization to organization. Also, it is not limited to organizations; it also includes application business logic because every application will have different logic and workflows.”

Me: “Alright, what would be the basic functionality test in our case?”

Sonia: “Since we are building a video consultation application, the basic functionality testing would be that the user can log in/sign up for the

application. Also, the video functionality should work fine. Only then can we say that it has cleared the smoke test.”

The smoke test can be seen in the following figure:

Figure 2.7: *Diagram representing smoke testing*

Me: “Nice! Any plans on how we are going to do this?”

Sonia: “Yes, so we will use selenium for automated smoke testing. It is a decent tool and has been used in industry for a long time. Also, it is reliable in terms of community support.”

Me: “Okay, got it. So, in our CI pipeline after the deployment, I have to execute selenium scripts for smoke testing?”

Sonia: “Correct. But make sure your CI server has the browser installed in headless mode.”

Me: “Sorry, I did not get the headless browser thing. Why do we need it?”

Sonia: “In simple terms, the headless browser is a web browser without any GUI. We need it because our application will open on different browsers, and we have to ensure that it runs perfectly on every browser. The reason behind keeping it headless is that we are doing automated testing, and GUI will become a hindrance in that.”

Me: “Well, you have planned everything. That is awesome! Thanks a lot for all the information. Okay, now I have to do some work, but I will sync with

you again for regression testing."

Note: Some Functionize customers report that they can uncover and fix as many as 80% of the bugs they discover simply by configuring and executing a solid smoke testing suite. This corresponds well with the Pareto principle of 80/20. For many teams, smoke tests might be covering only 20% or less of all test cases and yet catch 80% or more of the bugs. This alone makes smoke testing efforts worth the time investment.

Regression testing

That day, I got busy as I had quite a few tasks pending, and it took an entire day to resolve those. While I was riding back home, I was thinking about the discussion I needed to have with Sonia regarding regression testing. I was still guilty about forgetting post-deployment checks in my plan. Deciding to do something about task management, I planned to talk to Sonia the next morning.

Sonia: "Good morning!"

Me: "Hey, Good morning. I was about to come to you regarding regression testing. I would have come yesterday but got pulled away in some high-priority tasks."

Sonia: "Yes, I also got busy yesterday, but let us go to a conference room for that now. I can see that Room 3 is available."

Me: "Sure, lead the way. So, in regression testing, we verify that new code changes are not breaking anything in the existing codebase. The new changes should not influence the existing features of the product. Am I right?"

Sonia: "Right, in regression testing, we run the automated tests again to check the previously working functionality as well as the new functionalities. Typically, we do regression testing when new feature is introduced, when some of the bugs are fixed, during performance improvements, and during configuration level changes."

Me: "I have a question. Why do we need regression testing when we already have functional testing? I think functional testing also tests the functionality of each module."

Sonia: "Functional tests only inspect the behavior of new changes or features and do not check how much they are compatible with existing ones.

Therefore, without regression testing, it would be very difficult and time-consuming to identify the root cause of product failure."

Me: "Oh, I get it. Also, I have seen people include performance testing as a part of regression testing."

Sonia: "Yes, the rationale behind it is that there would be some changes that will impact the application performance, and no one wants a slow responsive application. So performance testing is required whenever a configuration or architecture level change is done, but I think that can be done manually."

Me: "I think we should do automated regression testing because it will save a lot of time. I mean looking at the facts, sometimes it can take around 6-7 hours for manual regression and performance testing. With automation, you and your team can use your time and energy at a better place, and the machines can handle this thing."

Sonia: "This makes sense to me because no one wants to stare at a screen, pushing buttons and waiting for the result. I would rather get the test results as a notification."

Regression testing is illustrated in the following figure:

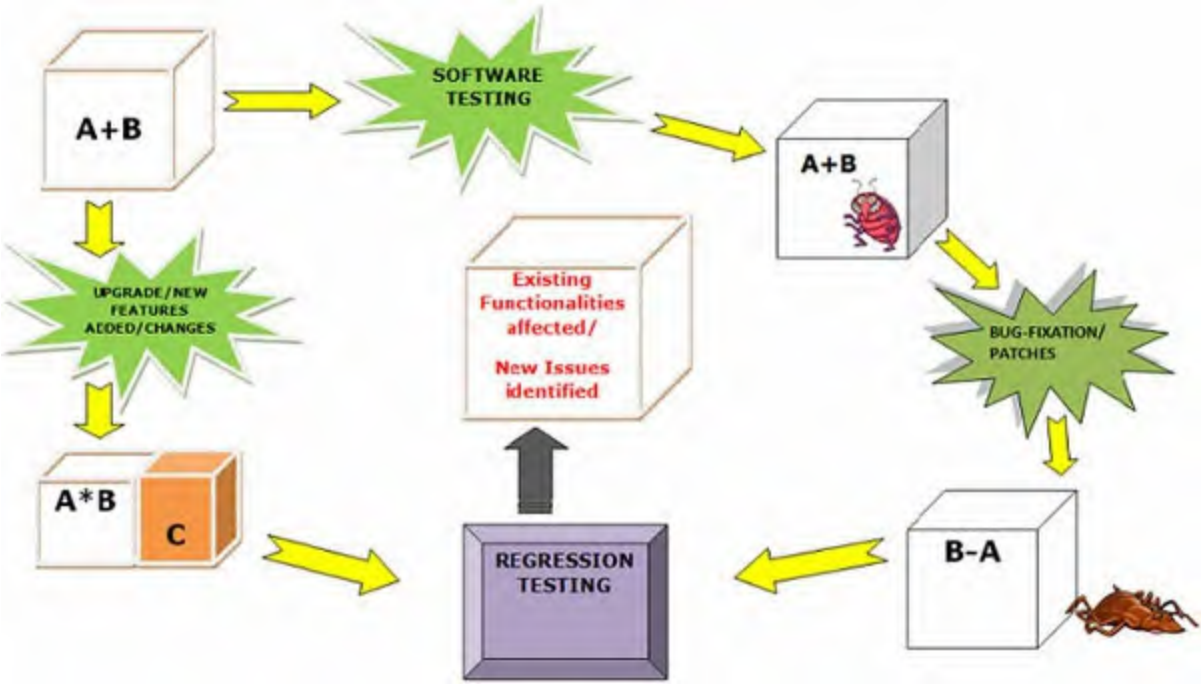

Figure 2.8: *Diagram representing regression testing*

Me: "That's the spirit! So, what tool are you going to use for regression

testing?”

Sonia: “I think Selenium would be sufficient, as our application is browser based and Selenium provides cross-browser regression testing. We should leverage it completely. Also, it’s not like we always have to run the regression test; it would be needed as per the release cycle.”

Me: “Okay, great. I have also thought about the performance testing part. Would you like to hear about it?”

Sonia: “Well, enlighten me.”

Me: “I am thinking of using JMeter for load testing. There are multiple tools available, like k6 and locust. The reason why I am opting for JMeter is that it is quite a mature tool and has great community support. Also, we can record our test results and compare those at every release cycle.”

Sonia: “Sounds good. But when are you planning to run all these?”

Me: “I will integrate it with our CI server so that it can be run with the regression suite itself.”

Sonia: “Well, if you do not mind, I’d suggest running regression and performance tests separate from the CI pipeline, maybe a nightly activity.”

Me: “Uhm, why is that?”

Sonia: “Well, you have smoke testing in your CI pipeline; that should be enough. Putting regression and performance tests as well will only increase the time and cost of the build. You see, these tests usually take a long time to run. If developers have to wait that long at each build, don’t you think it will defeat the purpose of CI?”

Me: “Hm, if you put it that way, I’d have to agree. We should consider that multiple developers may contribute their features and run builds multiple times a day.”

Sonia: “Glad I could help! This will reduce a lot of manual effort for me.”

Me: “Yes, it will.”

Fact: Facebook is known to be a pioneer in continuous deployments and had already achieved a cadence of 60,000 releases per day just for its Android app way back in 2017. That number is sure to have multiplied by now. The challenge with releasing continuously and at a frantic pace is to ensure that things don’t break because of releases. To this end, Facebook

gives careful thought to regression testing. It equips developers with tools to better gauge the impact of the code they write before it is released into production.

API testing

A few days later, I was having one of my quarterly fillings of water in the break room when Adeel and Sonia walked in. We greeted each other, and they started pushing the buttons on the coffee machine. I thought that was odd considering it was midafternoon. Also, I had never seen Adeel having coffee before. But most of all, they of them looked exhausted.

Me: “How’s everything going?”

Adeel: “Good, but Sonia and I had a rough night yesterday looking at a production issue.”

Me: “Oh, what happened?”

Sonia: “Our application had stopped the user registration. We debugged for hours in the front-end application, but in the end, we found out that the issue was in the back-end application.

Me: “So our back-end application is not tested properly?”

Sonia: “It is tested functionally, but there wasn’t validation for the API response. This is why we didn’t know when it was giving an empty response in cases where our front-end application needed a valid JSON structured response.”

Me: “Hmm, well if you all don’t mind, can I suggest something?”

Adeel: “Sure, go ahead.”

Me: “I think you should do API testing of your back-end code.”

Adeel: “You know that was in the plan; but if we do it, we will miss the delivery deadline.”

Me: “Maybe I can help you guys here. We are implementing the CI/CD pipeline, so we can add API testing as a part of the post-deployment check.”

Sonia: “Sounds nice, but again, it would be tedious for us.”

Me: “Yes, but I think it should be a one-time effort in which we will define the automated API test cases and execute them post-deployment. This way, your back-end API response will be thoroughly tested.”

Adeel: “Well if you both agree, I can talk to other team members regarding API testing.”

Me: “Yes, you should talk to them. Let me tell you some benefits to help convince them:

- We can find out the ways an end user may mess up by sending wrong data
- We can check the average response time of API to know if it is not too slow
- Request type validation, like request is POST, GET, or PUT can be passed while making a request
- We can check whether the request is authorized to make API call
- Also, we can do status code validation
- We can prevent SQL injection as well using these tests

The architecture of API testing looks as shown in the following figure:

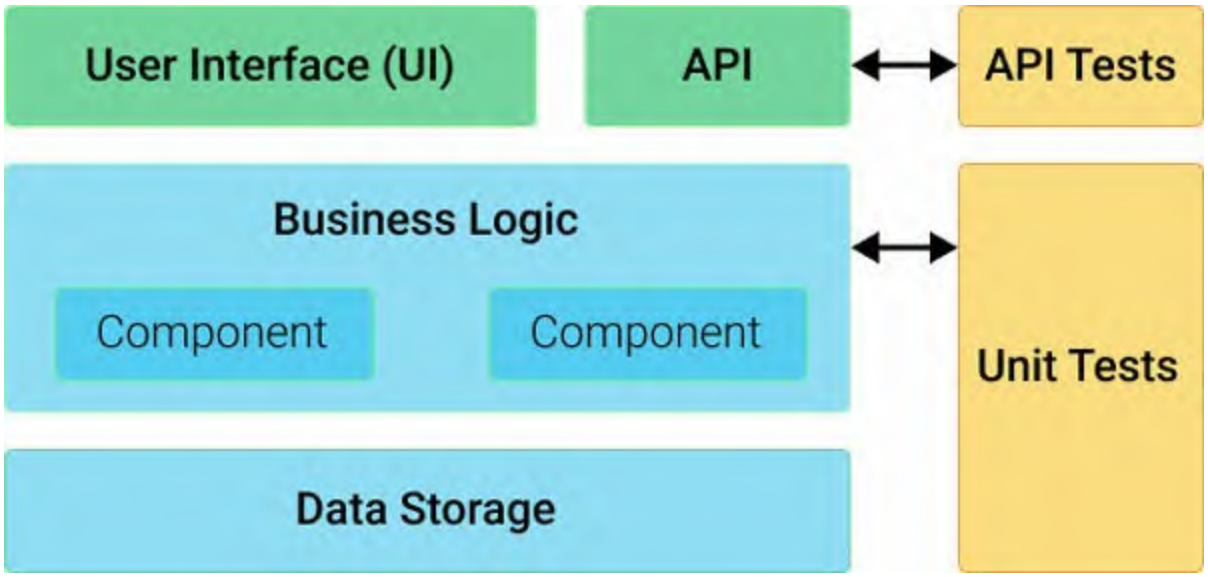

Figure 2.9: *Diagram representing API Testing*

Adeel: “Whoa! You have great knowledge of API testing.”

Me: “Yes, I had implemented this in my previous organization.”

Adeel: “That is great! You must know the tools for API testing as well then; I have heard about SOAP.”

Me: “Ahh, I was going to recommend the same. It is a good tool for API

testing and provides a headless interface as well."

Sonia: "Thanks a lot Abhishek! If we do this in time, I'll give everyone a treat."

Fact: According to Google Trends, the interest in API/web services testing has been growing steadily over the last couple of years. According to research by SmartBear, over 3,372 software professionals were part of API testing job profiles in 2019, and 91% of respondents have or plan to have a formal API testing process soon. About 45% of API testers reported that their organization automated more than 50% of test projects. Besides, API quality is a top priority for more than 75% of industry organizations.

Notifications

I had been doing satisfactory research for the past few days involving what to implement in the CI process to streamline the development as well as testing cycle, enhance productivity, and ensure better quality. While exploring these concepts, I made sure I utilized them to the best of my capabilities. Be it maintaining a code repository, including pre-build checks, or ensuring code quality, security enhancements, versioning artifacts, or database updates, I had a defined action plan. The layout was ready to be proposed in the next scrum meeting.

No sooner did I think to wrap up for the day than some concerns shown by Adeel crept into my mind. I remembered him asking, "It will be really great to have all the CI stages you mentioned in our application build process, but how can we make sure everything goes smoothly; if, say, there is an error at a stage, how would we get to know about that?". I repeated what Adeel said once again in my mind and thought that it was of vital importance to use notification systems to apprise the involved team members. While thinking of notifying, sending an e-mail notification came to my mind as the first thing that I could integrate with the CI process.

Although email notifications may get very detailed information, if we are already getting lots of emails, important alerts may get lost. So, I was exploring one more solution that would notify users instantly via text message, push notification, or an update on instant messenger. While exploring the other options, I discovered, to my surprise, that there could be dozens of options available just to serve the notification, which I can now

shortlist depending on what channel we are currently using.

During this research, I came across the term intelligent notification, which is crucial in terms of organizing when, how, and who should be notified. I landed upon the following options with me in the order of feasibility:

- Email
- Slack
- Pagerduty
- Skype
- Jabber
- HipChat

Branching strategy

Once all the CI checks were in place, the only remaining piece was the branching strategy that helps developers develop features rapidly. Bug fixing should also be easy in it. Again, I went to Adeel for a discussion.

Me: “Hi Adeel, our CI implementation is almost final, and I think it is pretty much streamlined. But I wanted to discuss a point that you had raised in earlier scrum meetings, regarding branching strategy.”

Adeel: “Yes, I had something in mind regarding this too, so we can discuss it.”

Me: “So Adeel, first of all, I would like to know what challenges you guys are facing right now.”

Adeel: “Well, that is a long list. But let me try to cover it in short. Right now, all developers are doing the development in a single branch, which is being used for all environments. So, we mostly spend a lot of time fixing merge conflicts. These are for all the development components.”

Me: “So how do you guys fix bugs?”

Adeel: “Sheesh, I am not very proud to tell you this, but we fix the bug in the same branch and then deploy the code from that branch.”

Me: “Wow, that is a mountain of problems right there.”

Adeel: “Tell me about it. We often cannot test new features because some other feature is still in progress. The release is a headache as well. And don’t

even get me started on rollbacks. It is almost impossible. I mean sure, we can cherry-pick and roll back certain commits, but it is not possible to do that on a regular basis."

Me: "Hmm, I think we need to have a proper branching strategy in place for this, otherwise it will keep affecting the development productivity."

Adeel: "Yes, I thought about implementing this many times but could not get the time to do it. But yes, if we can do this, it will reduce a lot of pain for developers."

Me: "So I have thought of a branching strategy. Let me share it with you, and then you can tell me if this is the right way to do it. I think there should be a development branch, and developers can create a feature branch from it. Once the code is reviewed and tested with the CI pipeline, we can merge that branch in the development branch, and from that development branch, we can create release branches to deploy code across different environments."

Adeel: "Okay, what will be the strategy for bugs?"

Me: "In case of bugs, a new hotfix branch will be created. From that, the release branch and fixes can be deployed from there."

Adeel: "According to me, we really don't need a complex branching strategy. Our projects are quite straightforward and don't have such complexities, so I think a simple approach would be much better. I think the branching strategy should be defined according to the project. We need a complex or nested branching structure if the requirements are complex, but I think our scenario is not that complex."

Me: "Do you have any strategy in mind?"

Adeel: "I will not change your idea completely because some of the parts are useful. We can have a development branch, and developers can create a feature branch from it once they are done with the feature. They will raise a Pull Request for the development branch, and then I can merge the code after review. Once the code is merged, the CI job will be executed to create and deploy the artifact. After the testing, a new Pull Request will be raised for the master branch, and once the Pull Request is merged, the code will be deployed in the production environment.

The developing branching strategy can be seen in the following figure:

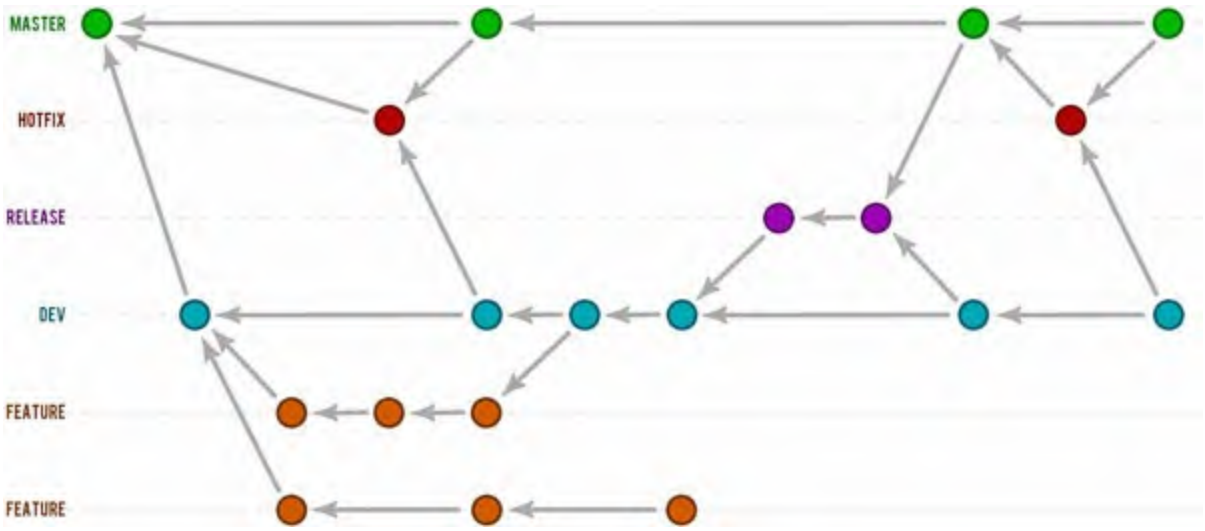

Figure 2.10: *Develop Branching Strategy*

Me: "Well, this sounds like a much simpler approach! But how will we fix bugs?"

Adeel: "So if there is a bug, a new hotfix branch will be created from the master branch, and it will be merged with the master after testing. The development branch and features branches will take a pull from it for the latest code."

Me: "Awesome! I think this will not create unnecessary confusion in terms of branching. We can follow this approach, for sure."

Adeel: "Yes, but if you have any feedback, you can tell me."

Me: "As far as I can think, this strategy looks fine to me."

Fact: Linus Torvalds strongly advocates for branching and merging as key practices that will greatly benefit software development.

Conclusion

As we still have the memories of the first sprint retrospective meeting alive in our minds, we were hoping for the best while walking to the meeting room for a mid-sprint sync-up with Sajal. There was a bit of nervousness, especially for me, as I need to present the concepts and feasibility around the fancy terms I talked about a week ago. But, to my surprise, the excitement was rushing ahead of anxiety as I started presenting quite a lot of CI concepts in front of him. With the development team and QA already satisfied after my

regular sync-up and discussion with them around the explored concepts and tools, convincing the rest became rather easily, maybe because the Developer and QA were mostly nodding throughout my session, giving me a boost of confidence.

During the discussion, we walked through everyone with the findings which revolved around:

- Pre-deployment checks, including code stability, code quality, unit tests, and code coverage and security tests
- Intermediate operations like artefact management and database versioning
- Post-deployment integrations with Smoke, Regression, and API tests
- Intelligent notification management
- Branching strategy overview

And finally, a week's worth of effort paid off after seeing an acknowledging response from Sajal as well. It ended on a positive note, as the team was discussing while returning to their desks.

In the next chapter, we will look at tools that the can be used for CI implementation, particularly Jenkins. We will also look at the comparison of different tools and dive a little deeper into the implementation of Jenkins.

Points to Remember

- Generally, people divide continuous integration into two parts: pre-deployment checks and post-deployment checks.
- DB versioning is an important aspect of Continuous Integration if we want to achieve a complete automated CI pipeline.
- Code coverage depends on the application test cases.

Multiple Choice Questions

1. **SonarQube is used to ensure the quality of code in the CI pipeline.**
    a. True
    b. False

2. **What type of testing do we perform to validate the web applications method?**
    a. Smoke testing
    b. Regression testing
    c. API testing
    d. None of the above

Answers

1. **a**
2. **c**

Questions

1. What is the branching strategy, and what is its importance?
2. What are the post-deployment integrations inside continuous integration?
3. What are the pre-deployment integrations inside continuous integration?
4. What are artifact management and DB versioning?

Key Terms

- Code quality
- Code stability
- Code coverage
- Unit testing
- Smoke testing
- Regression testing
- API testing
- Branching

CHAPTER 3

Introduction to Jenkins

The understanding of Continuous Integration was successful, and we have compared many tools. Different tools are available in the market with different types of functionalities and licensing. In this chapter, we will understand how we landed on Jenkins as the Continuous Integration tool and the different functionalities that Jenkins provides. Also, we will make a detailed comparison of the different CI tools.

Structure

The following topics will be discussed in this chapter:

- Tooling landscape for Continuous Integration
- Why Jenkins
- Jenkins installation on different platforms
- What are plugins, and why do we need them?
- Authentication and Authorization inside Jenkins
- Jenkins Pipeline
- Shared library
- What if the Server gets deleted
- Master-Slave Architecture
- Global tool configuration

Objectives

After studying this chapter, you should be able to decide whether to choose Jenkins as a CI/CD tool. You should also be able to configure different components of Jenkins and customize it to fit your needs, leverage the plugin model to add features enhancing productivity, write and maintain pipeline as code, take proper backup of the server, and scale server using Master-Slave

architecture.

Tooling landscape

“So, Monday, we meet again. We will never be friends, but maybe we can move past our mutual hostility towards a more-positive partnership.” And yes, why not, when it seems like a never-ending affair that revisits every 7th day? Like this, many stray thoughts were running through my mind during my not-so-comfortable travel to the office today. I couldn’t resist having some alternate means of commuting, as the current one is not worth my time and effort. Finally, I gave a sigh of relief as I stepped inside the office and took a couple of minutes to settle myself.

As usual, I took my morning dose of caffeine, self-prescribed, and while traveling back to my workstation, I almost drew a rough draft of the tasks in my mind, not noticing that my walking steps were getting longer and faster to preserve the latest volatile thoughts till I transferred them on paper.

Indeed, Continuous Integration doesn’t get rid of bugs, but it does make them easier to find and remove. To implement the CI, I need to look for a platform that will serve every aspect of our needs. In fact, as per my initial research, it should be very close to fitting the following needs:

- Minimum Setup Complexity
- Opensource
- Available Integrations
- Platform Agnostic
- Faster and Stable Build
- Build-As-Code

I can closely relate it to my daily commute problem where I was thinking of choosing an alternate option that could save my time and would be more comfortable than the current one. It’s learning from my life that I want to implement in my professional work to smoothen things and avoid future efforts to explore alternate technologies. So, the first thing to do today would be to list down all available options and choose the one that closely fulfills the requirements.

While summarizing my last week’s explorations, I was able to prepare a

high-level flow of multiple stages that will be required to implement the application build process using Continuous Integration:

- Developers check out code into their branches
- When done, the code will be merged to the deploy branch for that respective environment
- The CI tool would have a bunch of tasks:
    - Monitor the repository and check out changes when they occur
    - Build the application and run unit and integration tests
    - Release the deployable artifacts for testing
    - Assign a build label to the version of the code it just built
    - Inform the team of the successful build or failure at any of the previous stage
- The team will perform the fix, if needed
- Commit the changes to their branch and merge it to deploy branch

This way, the CI process will be running till we attain the success status for the component.

Available toolset

Throughout my research on the Continuous Integration process, I found that it adopts a more advanced programming practice to help developers prevent serious integration pitfalls.

As the journey continues to build the project, there are more things to integrate over time. To resolve the same, CI Tools automate many tedious tasks and make it easier to quickly identify potential issues before we end up releasing a disaster. There is a vast scope of tools primarily categorized as follows:

- Version Control System Integrated Pipelines
- Software as a Service
- Self-hosted and Managed

Let’s list some of the top CI Tools in each of the preceding categories. Of course, finally, we’d discuss the chosen one, comparing and listing the

reasons for selecting the “Swiss Knife.”

VCS Integrated Pipelines

Gitlab CI/CD

GitLab CI/CD is a tool built into GitLab for software development through the following:

- Continuous Integration (CI)
- Continuous Delivery (CD)
- Continuous Deployment (CD)

GitLab CI/CD is configured by a file called .gitlab-ci.yml, placed at the repository’s root. This file creates a pipeline, which runs for changes to the code in the repository. Pipelines consist of one or more stages that run in a particular order and can each contain one or more jobs that run in parallel. These jobs (or scripts) get executed by the GitLab Runner agent.

GitHub Actions

GitHub Actions automates **Software Development Life Cycle** (**SDLC**) workflows directly in the GitHub repository, where it stores code and collaborates via pull requests and issues.

We can write individual tasks, called actions, and combine them to create a custom workflow. Workflows are custom automated processes that are set up in the repository to do the following:

- Build
- Test
- Package
- Release, or deploy any code project on GitHub

With GitHub Actions, you can build end-to-end continuous integration (CI) and continuous deployment (CD) capabilities directly in the repository. GitHub Actions powers GitHub’s built-in continuous integration service.

Workflows run in Linux, macOS, Windows, and containers on GitHub-hosted machines called ‘runners.’ Alternatively, we can host our runners to

run workflows on machines we own or manage.

Bitbucket CI

Bitbucket Pipelines is CI/CD for Bitbucket Cloud, which is integrated with the UI and sits alongside the code repositories, making it easy to get the building, testing, and deploying code up and running based on a configuration file in the repository.

To set up Pipelines, we need to create and configure the **`bitbucket-pipelines.yml`** file in the root directory of the code repository. This file contains the build configuration. Using configuration-as-code means it is versioned and always in sync with the rest of the code.

Software as a Service

With the overflowing number of **Software as A Service** (**SaaS**) based CI/CD tools available in the market. I am listing out some most popular, which are used extensively among the teams:

- TravisCI
- CircleCI
- CodeFresh
- Codeship
- Shippable
- Wercker
- Azure DevOps
- AWS DevOps

Like most SaaS products, some of the most significant benefits of opting for SaaS-based CI/CD toolset are as follows:

- There is no hardware or software infrastructure to maintain.
- We need not worry about server maintenance or applying software updates/patches.
- They tend to be easy to set up.

But these points are only the advantages of using this toolset. In parallel, I'd

also mention the downsides to leveling the playing field:

- The cost of usage for a SaaS CI/CD solution may increase as the team gets larger.
- With the increasing scale in terms of services and frequency of usage, the cost of your CI/CD system could inflate dramatically.
- Furthermore, not all SaaS support all platforms, tools, and environments.

Self-Hosted CI/CD Tools

Until this point, I was convinced there are many attractive points in favor of a VCS and SaaS CI/CD service. Comparing it with a self-hosted solution, I couldn't help but notice one potential benefit of a self-hosted solution. It is the extensibility that others don't provide to the same extent.

In addition to extensibility, self-hosted solutions typically have fewer limitations on building configurations and concurrent build jobs. Moreover, the self-hosted services can be customized with plugins/extensions to enable functionalities that are not included "out of the box." Even without plugins, self-hosted CI/CD tools often support development platforms, languages, and testing frameworks more than many SaaS solutions.

Conversely, there are some potential downsides to a self-hosted system. Perhaps the biggest would be that we would require to manage the infrastructure, which includes applying software updates/patches. Also, unlike a SaaS solution, self-hosted systems may require a time-intensive process.

After spending some time exploring, I could build a comparison of available CI tools with a good ecosystem.

The following image shows a comparison of the available CI tools:

	Jenkins	TeamCity	Bamboo	Travis	Circle	Codeship
Pricing	Free	$299-$1999	$10-$800	$69-$489	$50-$3150	$75-$1500
Operating system	Windows, Linux, macOS, any Unix-like OS	Windows, Linux, macOS, Solaris, FreeBSD, and more	Windows, Linux, macOS, Solaris	Linux, macOS	Linux, iOS, Android	Windows, macOS
Hosting	On premise/ cloud	On premise	On premise/ Bitbucket as cloud	On premise/ cloud	Cloud	Cloud
Container support	✓	✓	✓	✓	✓	Yes for Pro version
Plugins	*****	****	**	****	***	****
Docs and support	Adequate	Good	Good	Poor	Good	Poor
Learning curve and usability	Easy	Medium	Medium	Easy	Easy	Easy
Use case	For big projects	For enterprise needs	For Atlassian integrations	For small projects and startups	For fast development and high budget	For any project

Figure 3.1: *A comparison of available CI tools*

Jenkins is an open-source project written in Java that runs on Windows, macOS, and other Unix-like operating systems. The first point of attraction in Jenkins is that it is open source. Apart from that, it is community-supported and primarily deployed on-premises, but it can also run-on cloud servers. Its integrations with Docker and Kubernetes take advantage of containers to roll out even more frequent releases.

Why Jenkins?

Like everything, there are pros and cons to all solutions, and with the vast amount of CI/CD tools available in the market, there's no such thing as "*one size fits all*." What matters is the flexibility and extensibility that will help in the customized formation of workflow that will meet the requirements. To reinforce my choice of Jenkins, I am listing the possibilities and features:

- **Easy Installation:** Jenkins is a platform-agnostic, self-contained Java-based program ready to run with packages for Windows, Mac OS, and Unix-like operating systems.
- **Easy Configuration:** It is easily set up and configured using its web interface, featuring error checks and a built-in help function.
- **Available Plugins:** There are over a thousand plugins available in the Update Center, integrating with every tool in the CI and CD toolchain.
- **Extensible:** It can be extended utilizing its plugin architecture, providing nearly endless possibilities for what it can do.
- **Easy Distribution:** Jenkins can efficiently distribute work across multiple machines for faster builds, tests, and deployments across multiple platforms.
- **Notification Support:** Integrations available to notify the build status on various communication channels.
- **Active community.** The Jenkins community provides a guided tour introducing the basics and advanced tutorials for more sophisticated use of the tool. They also hold an annual conference DevOps World | Jenkins World.
- **No expenses required:** Jenkins is an open-source resource backed by heavy community support.

Figure 3.2 explains the architecture of Jenkins plugins and integrations:

Figure 3.2: *A view of Jenkins plugins and integrations*

Jenkins installation

Jenkins is an open-source tool that is backed by a strong community, so it supports almost every kind of OS architecture, like macOS, Windows, and Linux.

In Linux OS, Jenkins packages are available for families like Debian, RedHat, and Arch. For the Debian and RedHat families, we can simply use their package manager, and for other Linux distributions, we can use the jar package.

The only dependency for installation is **Java.**

Installation on Linux (Debian)

Installation of Jenkins in Linux is straightforward, but for sake of simplicity, we have categorized the process into steps.

Assuming that Java is already installed on the system, we will first add the repository key of Jenkins to our system:

```
$ wget -q -O - http://pkg.jenkins-ci.org/debian/jenkins-
ci.org.key | sudo apt-key add -
```

When the key is added, we can add the repository to the system:

```
$ sudo sh -c 'echo deb http://pkg.jenkins-ci.org/debian-stable
binary/ > /etc/apt/sources.list.d/jenkins.list'
```

Update the packages repository of the Debian system:

```
$ sudo apt-get update
```

Install the Jenkins package using **`apt-get`** command:

```
$ sudo apt-get install Jenkins
```

Let us start and enable the Jenkins service on the system:

```
$ sudo systemctl start jenkins
$ sudo systemctl enable jenkins
```

In case firewall is being used, we may need to enable the **`Jenkins port 8080`** in the firewall:

```
$ sudo ufw allow 8080
```

Now we can access Jenkins at **http://server_ip:8080**.

Note: In this installation process, the latest stable version of Jenkins will be installed. If you want to install a different version of Jenkins, you may have to download that version manually from the Jenkins repo.

Installation on Windows

For installing Jenkins on the Windows system, we have to download the "**`.exe`**" package from Jenkins. The package can be downloaded from here. Follow the given steps to move forward:

1. Unzip the downloaded file as shown in the following figure:

Figure 3.3: *Downloaded exe file*

2. Click on the **`Next`** button, as shown in the following figure:

Figure 3.4: *Jenkins installation wizard*

Note: If you are installing on a Windows server, it might ask how you want to run this program: as a local process or as a service. Some may prefer running this as a service in Windows, but you can also run this as an individual process.

3. Click on the **`install`** button, as shown in the following figure:

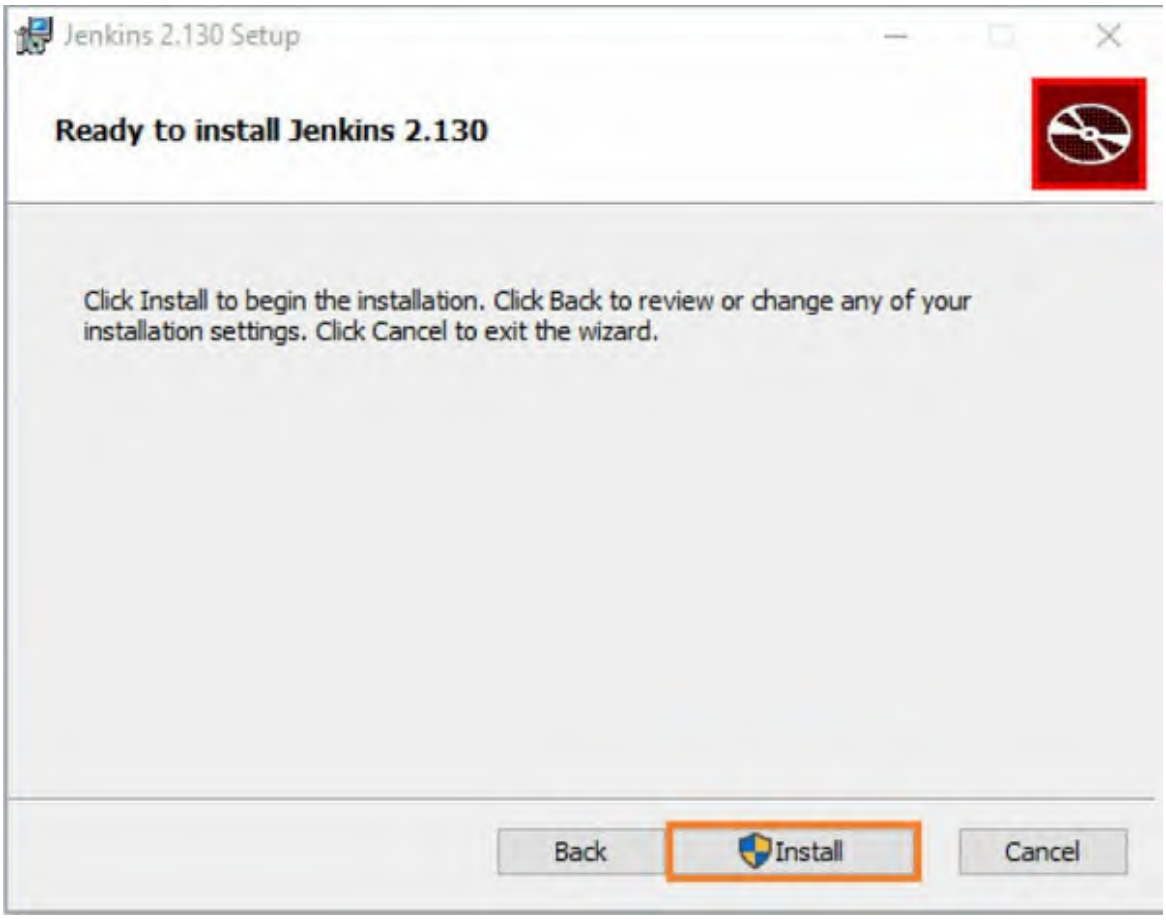

Figure 3.5: *Install Jenkins on Windows*

4. Once the installation is complete, you can open the Jenkins UI on the browser, as shown in the following figure:

Figure 3.6: *Unlocking Jenkins*

Note: In this installation process on Windows, we are using the 2.130 version of Jenkins; download the latest version for use. Also, if you are unable to find the path of the secret file, you can directly paste it into notepad explorer.

Ansible

If we don't want to follow these steps every time and want to install Jenkins in one go, we can use the ansible provisioner to install Jenkins.

There are multiple ansible roles available on the galaxy marketplace; here, we are providing a role for stable installation of the Jenkins.

```
$ ansible-galaxy install opstree_devops.jenkins
```

Create a "**hosts**" file in the directory where you have cloned this ansible role. Define the server connection details where Jenkins should be installed and configured.

```
[jenkins]
10.1.1.100 ansible_ssh_user=ubuntu ansible_ssh_private_key_file=
<file_path>
```

Create a file named **site.yml**:

```
---
- hosts: jenkins
  become: yes
  roles:
   -opstree_devops.jenkins
```

Once these two files are created, we can run the ansible by using this command:

```
$ ansible-playbook -i hosts site.yml
```

This ansible role is not limited to the installation part; it also helps in configuring Jenkins. Some features supported by this role are as follows:

- Plugin installation while installing Jenkins
- Credentials management
- Global tool configuration
- Default package installation

Plugins

One of the great advantages of Jenkins is that it is backed by a very strong open-source community, which develops thousands of useful plugins to enhance productivity. We can say that it follows the plug ‘n play architecture; if we want to add some functionality to our Jenkins server, we can simply add it by installing the plugin for that requirement.

Now the question arises: what is a Jenkins plugin and why do we need it? Well, let’s find out.

A plugin is an open-source tool that provides an extension to the feature pool of Jenkins. It enhances functionality and enables Jenkins to adapt to requirements.

Now that we are done with the “*what*” part, let’s talk about the “*why*” part. Suppose you have to integrate your current VCS system with Jenkins which builds a ‘Job’ every time a new commit is pushed to the repository. If we want to achieve this, we have to integrate the Jenkins server with our VCS system. For the integration part, we must write a custom library that will do this for us, but there are plugins already available to support this kind of scenario. Moreover, if the required plugin is not available at the Jenkins plugin center, we can create our plugin as well. The plugin development will be in Java.

Installation

Jenkins provides different methods to install plugins to the system:

- Using Web UI
- Using Jenkins-CLI (Command Line Interface)
- Using `.hpi` files (without internet connection)

In the first two approaches, Jenkins should be able to download the meta-information of the plugins from Update Center. Before discussing more about plugins, let’s understand what exactly Update Center is. It is a central repository where all Jenkins plugin information is stored and Jenkins connects to the Update center for downloading and installing the plugins.

The plugins get extracted from the `.hpi` file, which has all the required resources like code, dependencies, et. al.

Let us see how we can install the plugin using each method.

Web UI

This is the simplest and most used method to install plugins in the Jenkins Server. For installing the plugin from Web UI, go to **`Manage Jenkins`** > **`Manage Plugins`**.

Figure 3.7: *Plugin installation from Available tab*

Here, you will see there are multiple tabs available inside **Manage Plugins**:

- **Updates**: This lists all plugins that are already installed in the system but can be upgraded to the latest version. This meta-information is fetched from the Update center.
- **Available**: This lists all plugins that are not installed in the Jenkins Server but can be installed, for example, we have searched the "**`golang`**" plugin for installation.
- **Installed**: This lists all plugins that are currently installed in Jenkins.
- **Advanced**: The advanced option is available to use a custom repository like Update Center. Also, we can install plugins by uploading the **`.hpi`** file in this option.

At the bottom, we have three more options:

- **`Install without restart`**: This option installs the selected plugins but does not restart the Jenkins service.
- **`Download now and install after the restart`**: This option downloads the plugin but does not install it until the Jenkins service is

restarted.

- **`Check now`**: This option updates the version meta-information from the Update Center.

As per the best practice and our recommendations, you should always restart the Jenkins service after the plugin installation because sometimes, plugins don't work properly unless the service is restarted.

Jenkins-CLI

Jenkins also provides the flexibility to install and manage plugins using CLI because most DevOps or Sys Engineers like command-line interface for scripting and automation. For installing plugins from CLI, first, we need to download the '**`jenkins-cli.jar`**' file.

https://jenkins_url/jnlpJars/jenkins-cli.jar

> **Note:** Replace the dummy URL <jenkins_url> with the actual Jenkins server URL. It could be an IP or domain.

```
$ java -jar jenkins-cli.jar -s http://<your_jenkins_server -auth
<user:password> install-plugin golang
```

So, this is an example command to install a "golang" plugin in the Jenkins system using CLI.

The installation behavior can also be controlled by CLI, for example, if we want to install the plugins without restarting Jenkins service, we can use the "**`-deploy`**" flag; if we want to install with restart, we can use the "**`-restart`**" flag.

More options can be explored by the "**`-help`**" flag.

HPI Files (Without internet)

One more advanced way to install the Jenkins plugin is by downloading the plugin ".hpi" file and then installing it using the advanced **Manage Plugins** option. Generally, in gaming platforms, the game engine uses this format to store compressed data. This format is known as the Hemera Technologies Proprietary Image format. Jenkins also uses the same extension name for plugins.

https://updates.jenkins-ci.org/download/plugins

The "**.hpi**" files can be downloaded from the preceding link. Once the file is downloaded, go to **Manage Jenkins** > **Manage Plugins.** In there, click on the **Advanced** tab and upload the downloaded file. Refer to the following figure:

Figure 3.8: *Plugin installation from the Advanced tab*

You can also copy the **.hpi** file to the Jenkins plugins directory to install it.

Fact: Jenkins has 1500+ community-contributed plugins to support Jenkins build and automation. Jenkins was founded on 2nd February 2011 by its original author Kohsuke Kawaguchi.

Simple Plugins

Since we cannot list the thousands of plugins contributed by a strong community to the Jenkins Plugin pool, we will cover some of the commonly used plugins that are very useful as well.

Source code management (Git)

This plugin is kind of a backbone of the Jenkins server because it helps to integrate it with different VCS providers like GitHub and Gitlab.

The plugin URL is **https://plugins.jenkins.io/git/**.

Git plugins provide git functions for Jenkins projects. In addition, it provides some other features like poll-scm, and Pull Requests reviewer, as shown in the following figure:

Figure 3.9: *Git plugin*

In the git plugin, we can define the repository URL whether ssh and HTTPS. If the repository is protected, we can provide the credentials for the same. Also, we can select specific branches as parameters and execute 'Jobs' from them.

User Interface

User interface plugins help make Jenkins management cleaner and more convenient. These are a few plugins that can be used under the user interface:

- Folders
- BlueOcean

Folders

Folder plugin allows us to manage different Jenkins jobs in a hierarchical manner. You can co-relate this with the directory structure you use to manage files in your system.

https://plugins.jenkins.io/cloudbees-folder/

Let's suppose there are multiple teams or environments in an infrastructure. If we have all the 'Jobs' in a single view, it is not practical as it would be quite difficult to search and identify them. To overcome issues like this, we can have a separate folder structure in which we can move the job corresponding

to an environment or a team. If we need further refactoring, we can create nested folders as well.

You can create the Folder by clicking on **`New Item`** and then selecting the Folder option, as shown in the following figure:

Figure 3.10: *Jenkins project resources*

Here, we have a project folder named "**`Obvis`**", and we have managed all the jobs related to it inside that folder:

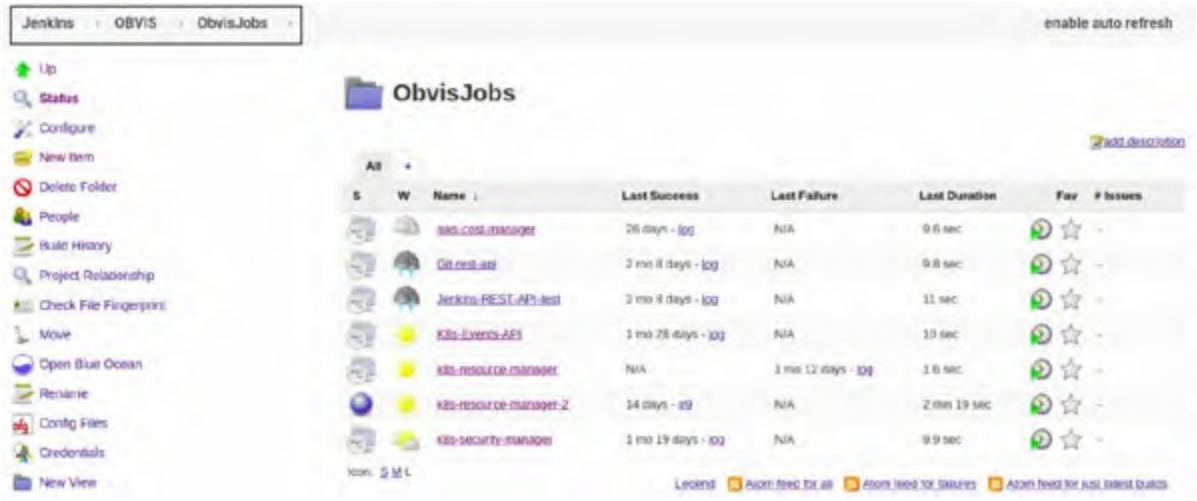

Figure 3.11: *Jenkins folder*

BlueOcean

BlueOcean is a plugin that provides a beautiful and elegant UI to Jenkins. It removes the clutter and provides clarity to each step of the Job.

https://plugins.jenkins.io/blueocean/

Some of the key features of the BlueOcean plugin are as follows:

- **Advance Pipeline visualization**: In simple pipeline UI, we have to scroll thousands of lines to see the output of a particular command, but that is not the case with BlueOcean because it provides a modular view for every step of the pipeline.
- **Branch and Pull Request Trigger Status**: If the VCS is integrated with Jenkins, then it can let you know the status of the branch or pull request as in how stable it is.
- **Precision**: This plugin can tell exactly what steps the pipeline broke.

The view of the BlueOcean stage is shown in the following figure:

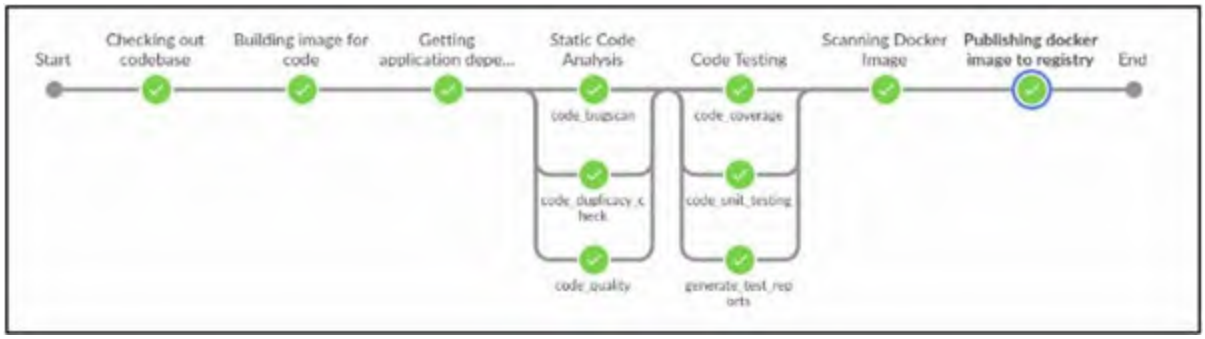

Figure 3.12: *Blue Ocean pipeline view*

Administration

Administration plugins are used to manage Jenkins's functionality. Similar to User Interface, there are multiple plugins to administer Jenkins:

- Agents
- AuditTrail

Agents

Agents is not exactly a single plugin. There's a group of plugins that we call agents. They are used to manage and connect to different types of agents like

Linux and Windows.

- **https://plugins.jenkins.io/ssh-slaves/**
- **https://plugins.jenkins.io/windows-slaves/**

So, the ssh-slaves plugin allows you to manage Linux machines over SSH. The ssh-slave plugin connects to the machine over ssh and then uses the remoting.jar file to manage that machine as a Jenkins slave, as shown in the following figure:

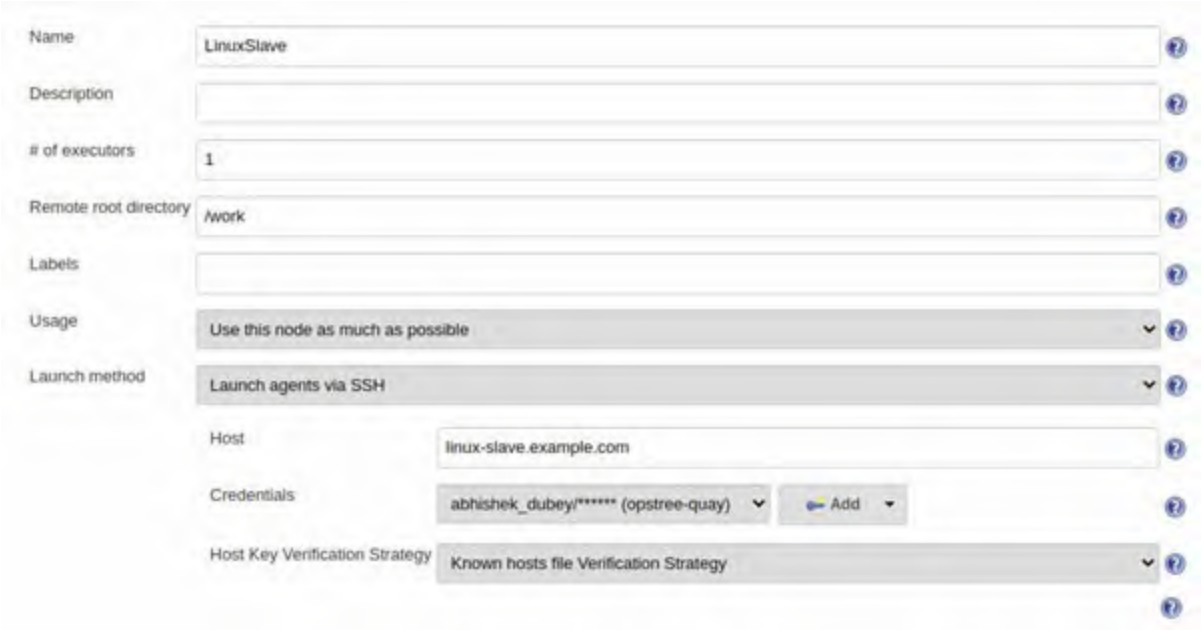

Figure 3.13: *Jenkins agents - Linux*

Similar to ssh-slaves, the windows-slaves plugin is used to manage Windows machines over the Remote Communication Service. Once the connection is established, it uses the remoting.jar to register the machine as a Windows slave, as shown in the following figure:

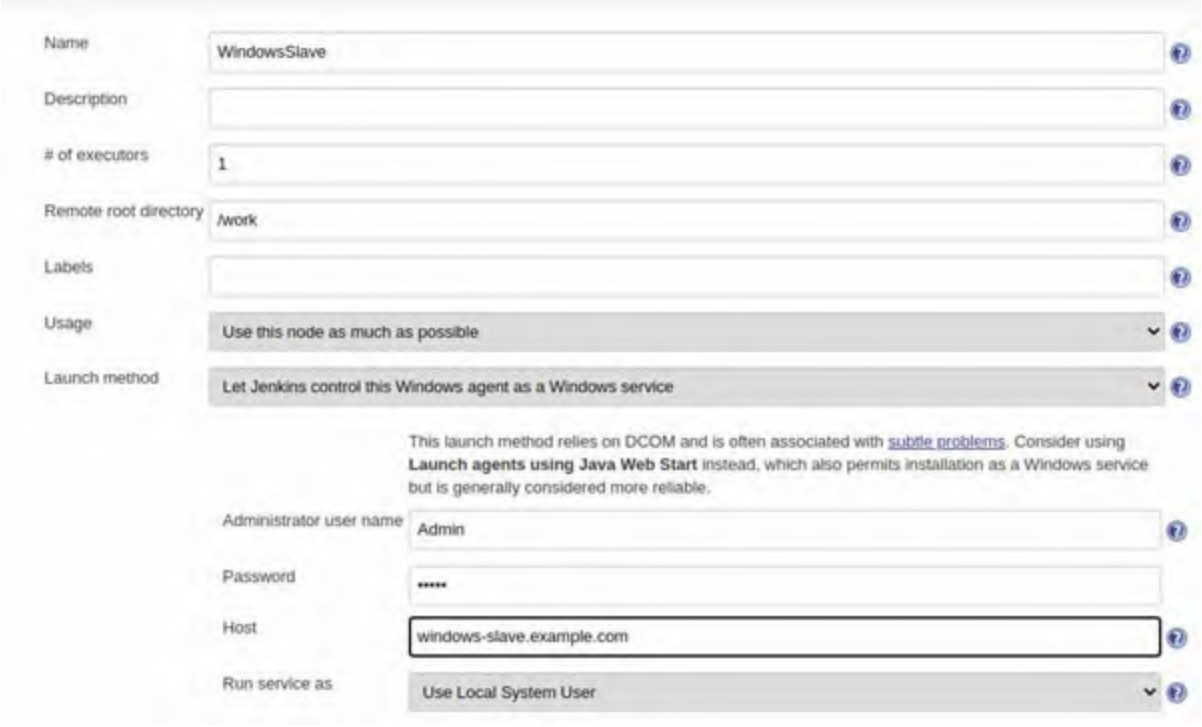

Figure 3.14: *Jenkins agent - Windows*

Note: We will discuss Jenkins slaves in depth under the Master/Agent topic.

Audit Trails

Audit Trails plugin keeps track of user actions, so it keeps the logs and events for every user who performed any Jenkins-related activities or operations. The log file location is configurable and will be written in the filesystem. The audit trail plugin is shown in the following figure:

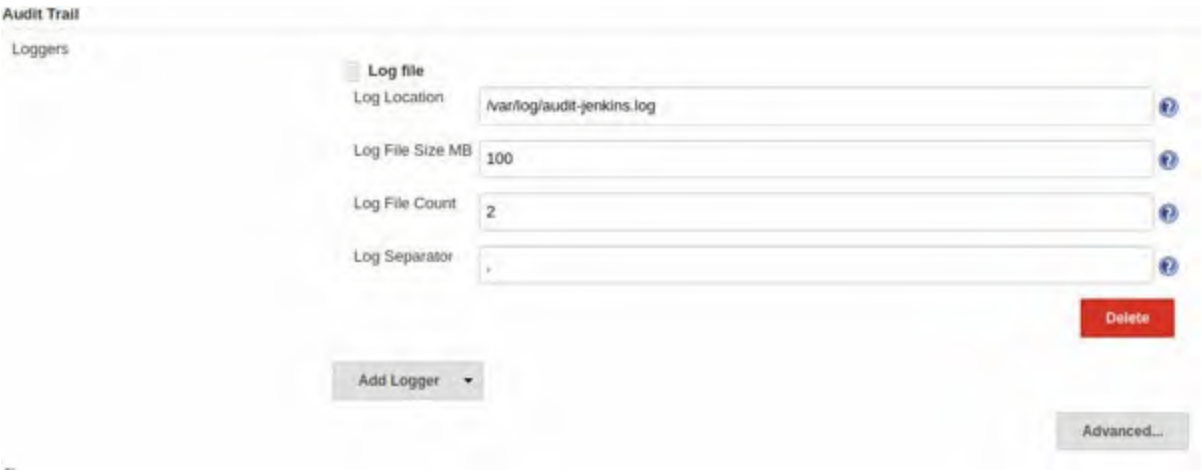

Figure 3.15: *Audit trail plugin*

Build Management

The Build Management plugins help us to enhance the functionality of the Jenkins job by providing features like reporting and quality gates.

Some build-management plugins are listed here:

- Warnings Next Generation
- HTML Publisher
- SonarScanner

Warnings Next Generation

Warnings next-generation plugins collect the warnings and issues from different compiler projects like C, and Java, and visualize them as results.

We can also say that this is a Static Analysis Suite for Jenkins that provides reporting for tools Android Lint, Checkstyle, PMD, Findbugs, and such.

In this plugin, we can also configure the rules for our Projects, and if one of the rules is failed, it will mark the build status as unstable or failed as per our choice.

The report view of the warnings next generation plugin is shown in the following figure:

Figure 3.16: *Warnings Next Generation plugin charts*

HTML Publisher

The HTML publisher plugin helps us to publish the HTML reports that are

generated by our build or Jobs.

https://plugins.jenkins.io/htmlpublisher/

A sample HTML report published by the HTML publisher plugin is shown as follows:

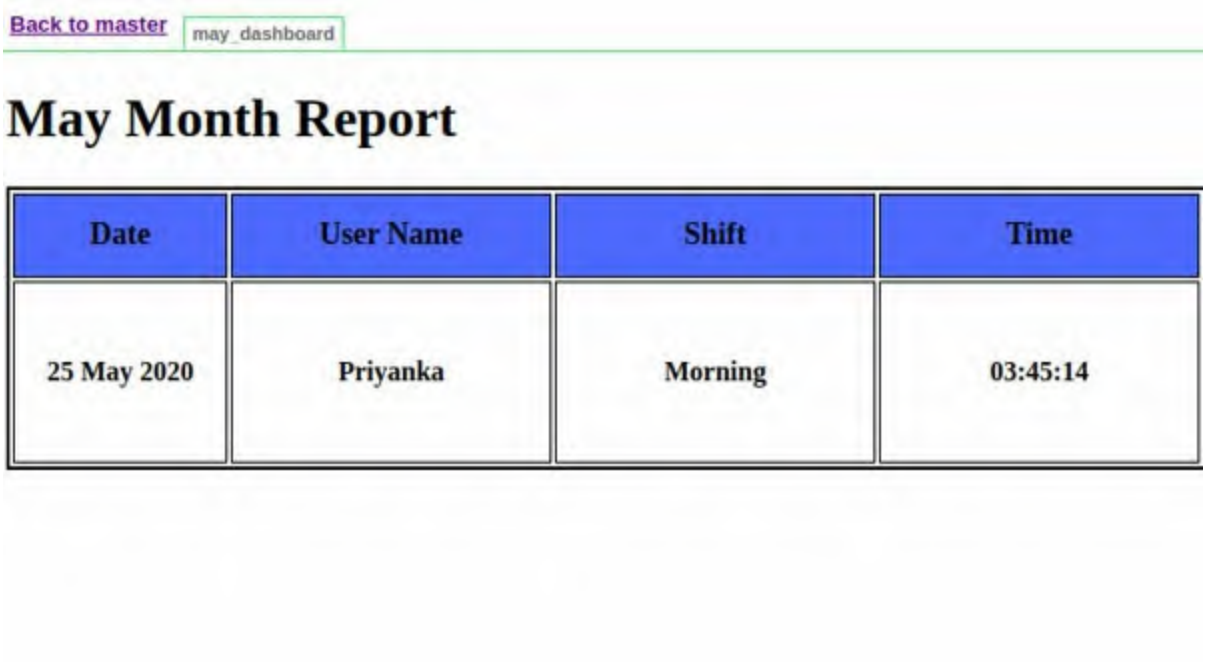

Figure 3.17: *Report by HTML Publisher plugin*

SonarScanner

The SonarScanner plugin allows us to centrally manage the Sonarqube connection details in Jenkins.

https://plugins.jenkins.io/sonar/

In addition to the connection with Sonarqube, it provides functionalities like integration with different build tools, i.e., Maven, Gradle, and MSBuild. With the help of this plugin, we can manage the sonar configuration of the project, which also means we do not need to change our Maven and Gradle-based projects at all. Refer to the following figure:

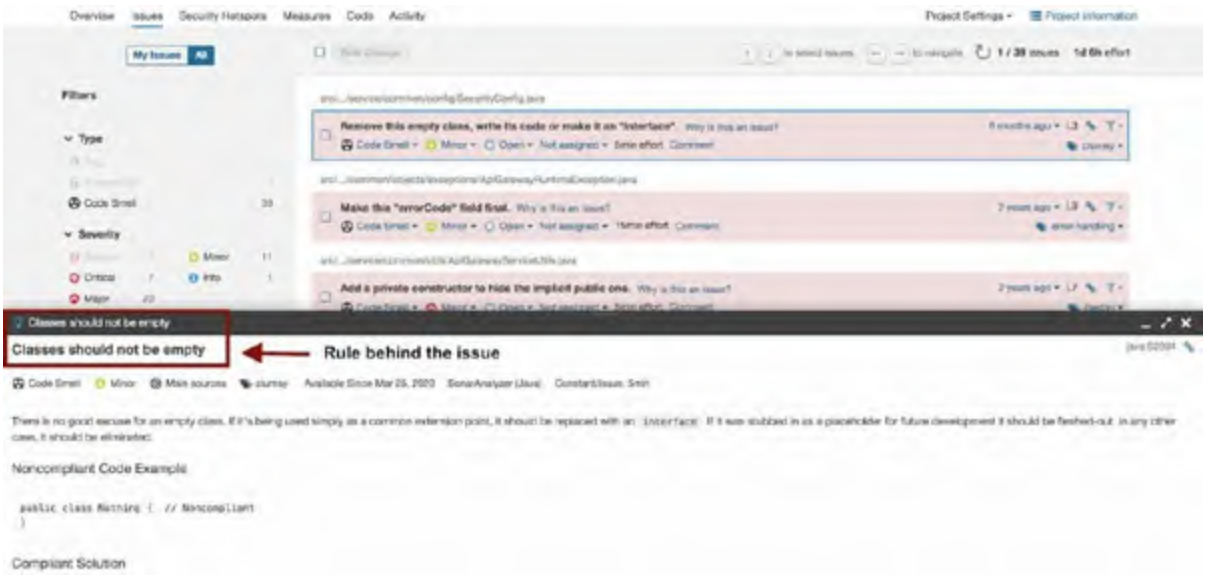

Figure 3.18: *SonarScanner plugin report*

Notification

The notification-related plugins are used to manage notifications with tools like Slack, Email, Hipchat, Pagerduty, and so on.

Since there are multiple clients of notifications, we will talk about the most popular ones:

- Slack
- Email
- Microsoft Teams

Slack

The slack plugin provides the flexibility to send notifications to different slack channels, and a similar plugin can be used for Mattermost and RocketChat.

https://plugins.jenkins.io/slack/

With the help of this plugin, you can send notifications to slack channels for events like job failure or new version release alerts.

A sample notification alert on slack is shown in the following figure:

Figure 3.19: *An example slack notification*

Email

The Email plugin is for sending email notifications; it also allows us to customize the behavior of emails: who should receive and what would be the email content. We can also add our custom email template.

https://plugins.jenkins.io/email-ext/

We can configure it on the global project level as well as individual project level. Through this plugin, we can notify the stakeholders or developers of their project events.

A sample notification alert on email is shown in the following figure:

Figure 3.20: *An example mail notification*

Authentication and authorization

Much like the security in the physical world, we need to consider it for our Jenkins server as well. Jenkins is a crucial part of our SDLC and, if careless, can become a gaping vulnerability for potential security attacks. It sits right in between our infrastructure from where one can have access to our codebase, registries, artifactories, application servers, and whatnot. Not only

that, but it is also a storehouse for passwords, access tokens, keys, and everything you know to be private.

Securing Jenkins has two domains: authentication and authorization. One means locking the main gate to your house, and the other means locking the rooms, wardrobes, drawers, and so on. Makes sense, doesn't it? You do want everybody involved in build and release to be able to log in to Jenkins, which is authentication, but you don't want them to go all Superman in there, so you want to restrict what they can do as per their role. It is not only valid for people. It goes the same for everything capable of executing commands. We'll explore this later.

Authentication

First, let us look at what types of authentication mechanisms is supported by Jenkins. Yes, it supports multiple authentication mechanisms, in case you were not aware. We call it the **Security Realm**. By we, I mean the Jenkins community. In the security realm, we have **Basic Auth** to start with, and then there are third-party integrations like LDAP, Active Directory (AD), Github Auth, SSO, and so on:

1. **Basic Security Setup** is the method where Jenkins maintains its own user database. It can be enabled from **`Manage Jenkins`** > **`Configure Global Security`**.

   Under Security Realm, click on Jenkins's own user database. Usually, the admin would control all user signups, so don't click on 'Allow users to sign up'.

   This method is simple and efficient but is not recommended for large infrastructures
2. **Delegate to Servlet container** is an authentication method where, as the name suggests, authentication is delegated to an external servlet, like Tomcat Server, that is hosting Jenkins server. In this case, we can configure users present in **`tomcat-user.xml`** to be allowed to log in to Jenkins. Refer to the following figure:

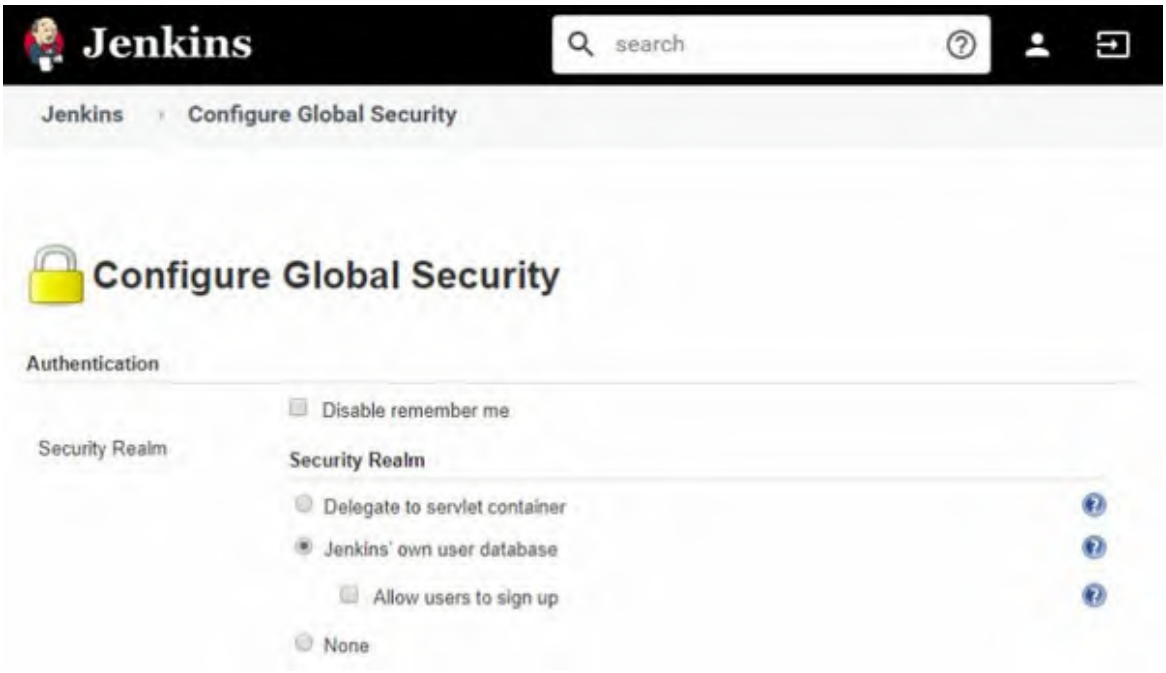

Figure 3.21: *Jenkins own user database*

3. **LDAP** requires us to configure an external LDAP authentication server. All users and groups will be authenticated through that server. Of course, we'd need to install and configure the LDAP plugin first. LDAP configuration settings are available in the same place, **`Manage Jenkins`** > **`Configure Global Security`**, after the plugin is installed. We can test connections to validate configuration by clicking on the '**`Test LDAP settings`**' button.
4. **Unix user/group database** in Linux OS can also be used as one of the authentication methods. It is easily configurable and can be used where one does not want to manage multiple user databases.
5. **Other Identity Provider Plugins** are also available apart from the authentication methods already discussed. There are several ways to manage Jenkins users through various identity provider plugins like Active directory, Google Login, and GitHub Authentication. They all have their own user databases. All we need to do is install their plugins, provide configuration details in **`Manage Jenkins`** > **`Configure Global Security`**, and use their database to authenticate users. This is also known as **Single Sign On** (**SSO**). SSO is widely used in many large organizations. It has many benefits. For starters, you don't need to manage user information at any level. That's one less thing on your

plate when managing large infrastructures. Other benefits include the following:

a. Users don’t need to juggle multiple credentials
b. It’s highly secure across devices
c. It reduces complexity in various aspects, like troubleshooting, information retention, and so on.

In *figure 3.22*, SSO example with Google authentication is shown:

Figure 3.22: *Using SSO for Authentication*

Authorization

Just like authentication, Jenkins supports various authorization methods as well. The first two, “**Anyone can do anything**” and “**Legacy mode**,” are not recommended for use in any setup but test servers. The first one is self-explanatory. In the second one, though, users with the “admin” role have all access, and barring those, all others have read-only access.

Apart from the preceding two, we have these methods of authorization:

- **Logged-in users can do anything** is useful for forcing users to log in before taking any action. This way, we can monitor their activity. However, we would not be able to restrict unnecessary access. It is configurable in the “Manage Jenkins > Configure Global Security” Authorization section.
- **Matrix-based security/Project-based Matrix authorization** is one of the most promising authorization strategies as it provides granular control over user access. After we have added users and roles in our Jenkins server, we can enable this authorization strategy and provide access to specific users on the tasks relevant to their job roles only. It is imperative to mention here that Matrix and Project-based Matrix

authorization are not the same. The latter is an extension of the former with some additional ACLs providing separate access control for different projects rather than for all projects at once. Both are available with the **Matrix Authorization Strategy** plugin. Refer to the following figure:

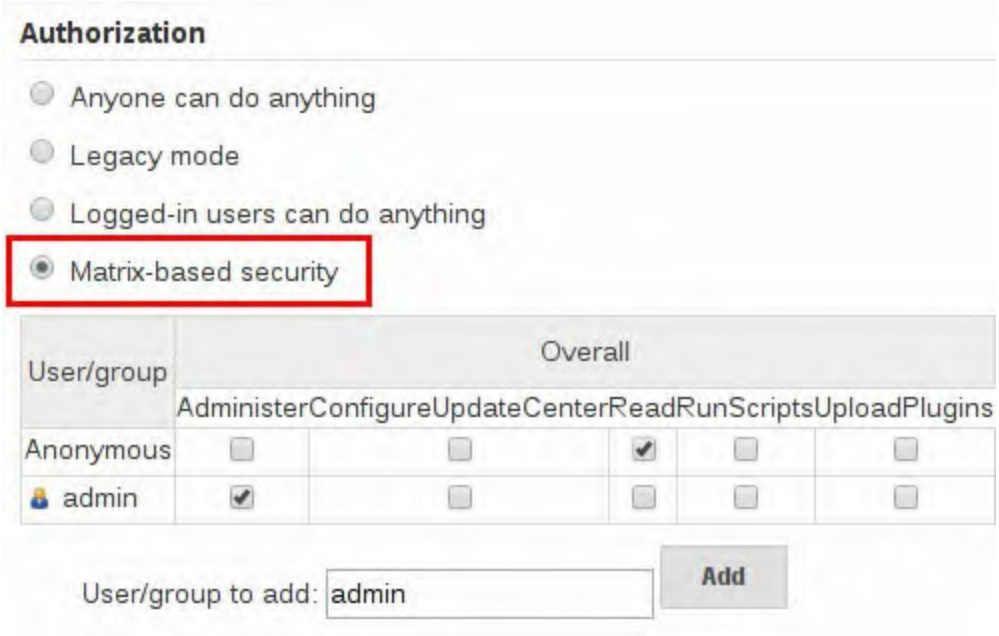

Figure 3.23: *Matrix-based Authorization*

- **Role-based Authorization** is a different plugin that provides everything that matrix-based authorization provides, with the exception of managing roles. With the help of this plugin, we can create and assign roles to users depending on their job description and then allow access in a matrix-based fashion to those roles rather than users. Of course, one role can have multiple users, and one user can be associated with multiple roles, in which case permissions from all roles will be added for the said user. You can manage roles from **`Manage Jenkins`** > **`Manage and Assign roles`**.

  The view of the panel where “Manage and Assign Roles” is available is shown in the following figure:

Figure 3.24: *Manage and assign roles*

Different types of available authorization modes are shown in the following figure:

Figure 3.25: *Role base strategy*

Recommendation

Having gone through all the information mentioned earlier, it is not difficult to choose the most appropriate set of authentication and authorization strategies for a production setup. It is, of course, SSO+Role-based authorization. Even if we put aside all the other facts, we cannot ignore that we like cleanliness. Even those who don't mind being cluttered would accept that they prefer things organized. When we have a large enough infrastructure, there is a lot of information going around, and we need to manage and organize everything to stay competitive and ensure productivity. SSO will give us one stop to manage authentication for all users. We don't need extra resources to manage user login information and security. And with a role-based strategy, we can create roles, both globally and project-wise, and

assign them to appropriate users. Then, assign access based on those groups rather than each user individually, forming an organized, stacked structure.

Jenkins Pipeline

Jenkins pipeline is not different than a normal pipeline installed in our houses and localities. Just like a normal pipeline, it also has a flow.

So, if we talk about the "*pipeline*" term in tech definition, it is a series of data processing or execution where the output of the next step depends on the input of the previous step.

Although pipelines are sequential in practice, the Jenkins pipeline concept supports the parallel execution of steps as well.

Let us see why we need Jenkins pipeline:

- Jenkins pipeline provides the functionality to write your pipeline/workflow in the form of code, which can be executed from any VCS.
- Since the job code is maintained over VCS, multiple users can perform development without affecting the running pipeline. In case of failure, we can revert to the previous pipeline easily.
- We can pause the pipeline in between for steps like approval.
- Jenkins pipeline can also integrate with blue-ocean automatically, which provides a great visualization of pipeline jobs.

In simple words, we can say that the Jenkins pipeline is a set of steps that get executed to achieve a common workflow. This workflow could be a continuous integration, continuous delivery/deployment, or any other task you want to automate.

The code or definition of the pipeline generally gets written in a file called "Jenkinsfile". The language in which we write the Jenkinsfile is "Groovy". It supports Groovy because it is a scripting language based on Java, and Jenkins is built on Java as well.

A few advantages of using Jenkinsfile are as follows:

- Jenkins pipeline can be automatically created from the Jenkinsfile in VCS, which means we can create a pipeline for all branches and Pull Requests.

- Jenkinsfile is a single source of truth for all the changes and execution so that you can track the changes of your pipeline workflow as well.
- It is extensible as well, which means if there is a requirement for your pipeline, you can write the custom libraries for it.

Note: Although we can write the pipeline in Jenkins UI as well, it is considered a best practice to write pipeline code inside the Jenkinsfile and then commit it to the Version control system.

For writing the Jenkins pipeline, we have to install two plugins in the Jenkins server:

https://plugins.jenkins.io/workflow-aggregator/

https://plugins.jenkins.io/pipeline-utility-steps/

Scripted vs Declarative Pipeline

A pipeline can be written in two types of syntax or method: scripted and declarative.

Scripted Pipeline: This is a traditional way of writing code for Jenkinsfile or Jenkins pipeline. It uses strictly groovy-based syntax. In this way of writing the pipeline, we have a lot of control over the pipeline script, and we can mutate it as per our use. This pipeline helps to develop complex features in pipeline code.

Here's an example:

```
node {
   stage("Hello World Stage") {
     echo "Hello World"
  }
}
```

Declarative Pipeline: Declarative pipeline is a new feature of Jenkins that provides user-friendly syntax to develop Jenkins pipeline. Writing and reading pipeline code in this syntax is easier than in the scripted pipeline.

The following is an example:

```
pipeline {
  agent any
  stages {
    stage ("Hello World Stage") {
      steps {
```

```
        echo “Hello World”
      }
    }
  }
}
```

For writing pipelines, we can use the Pipeline Syntax Generator option provided by Jenkins, as shown in the following figure:

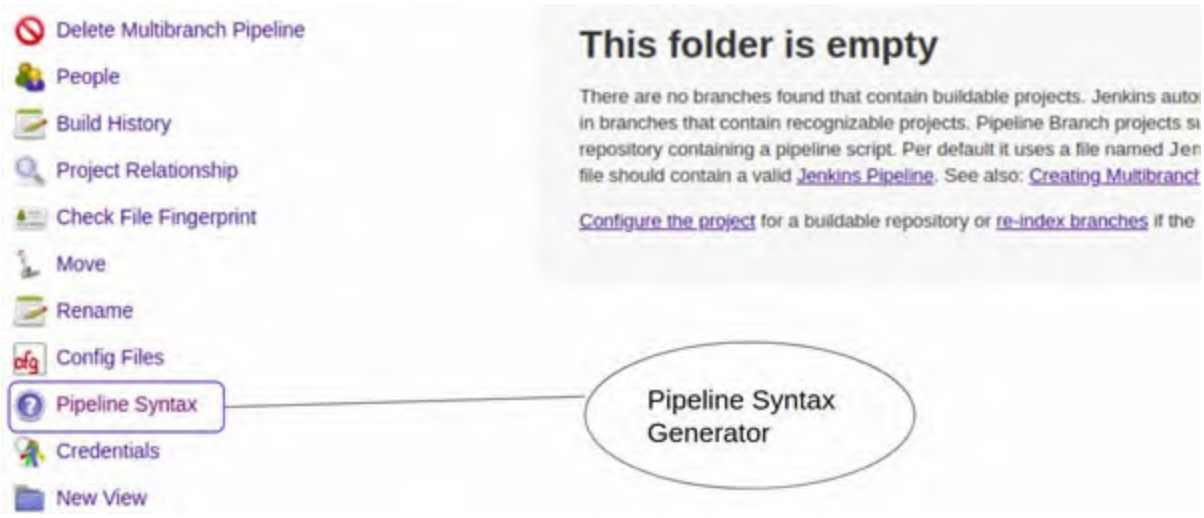

Figure 3.26: *Pipeline Syntax generator*

Once we click on this “**`Pipeline Syntax`**” button, it will redirect us to a page where we can generate the pipeline syntaxes. Consider this example:

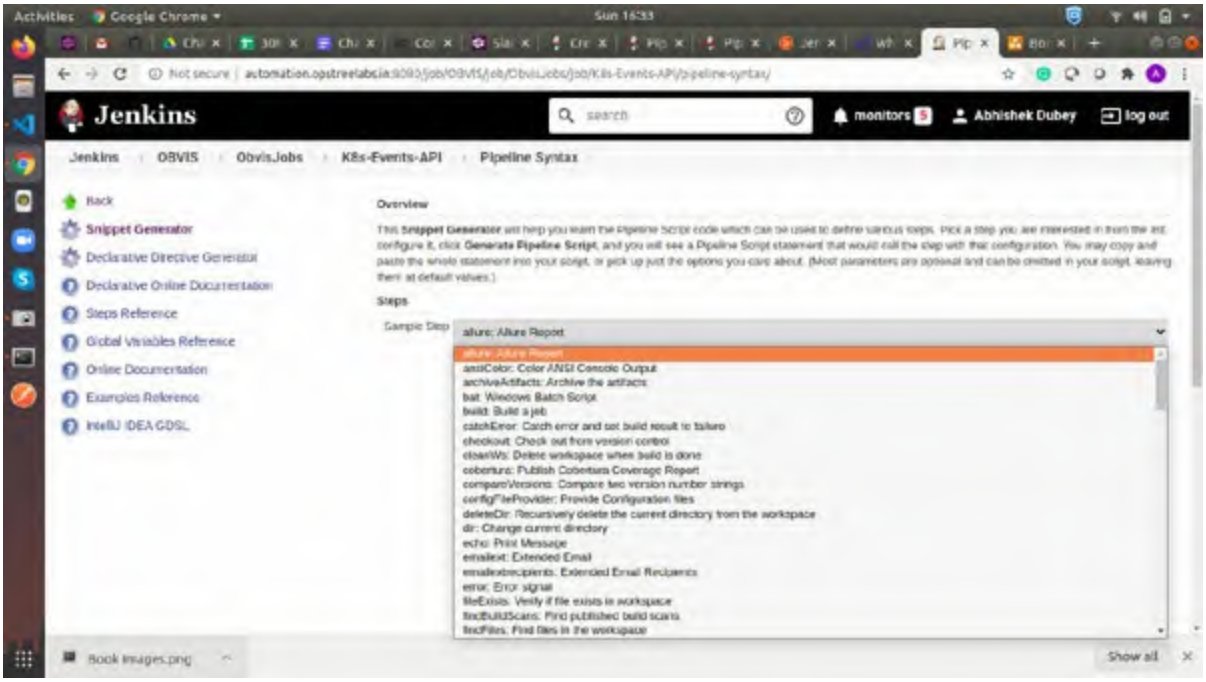

Figure 3.27: *Snippet generator drop-down*

Terms

We have multiple terms in the pipeline of Jenkins. While we cannot cover them all, we can cover the interesting and important ones. So here are a few important terms that are commonly used in the pipeline:

- Pipeline
- Node
- Stage
- Steps
- Parallel

Pipeline

Pipeline is a block in which we define the complete execution process, which typically includes information like the following:

- Which node to use for workflow/pipeline execution
- Different stages' information

Note: Pipeline term is a part of Declarative syntax only; we cannot use this in the scripted pipeline.

Here's an example:

```
pipeline {
  agent any
}
```

Node

Node block is used to define the host where Jenkins pipeline/workflow will be executed. By default, it is "master" but can be set to any other host as well.

Note: Node term is supported in the scripted pipeline only.

Consider this example:

```
node {
  echo "Hello World"
```

```
}
```

Stage

A stage is a sequence of steps in a pipeline. It helps in visualizing the pipeline in a better way. An example could be as follows:

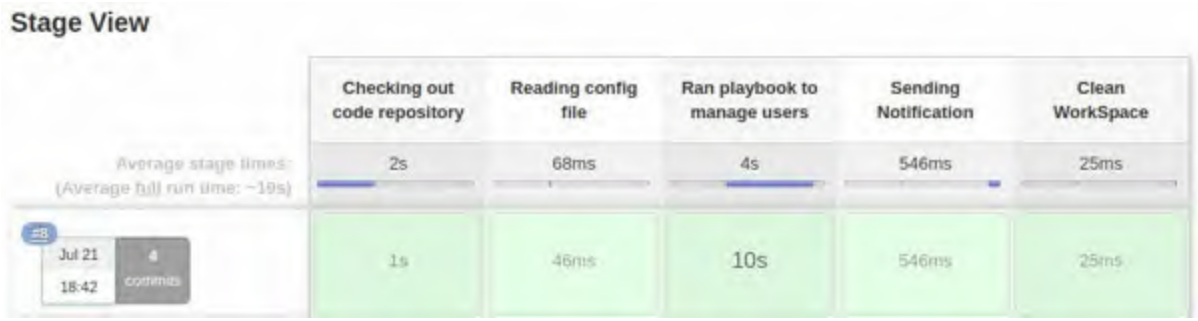

Figure 3.28: *Pipeline stages*

A simple example of pipeline stage in scripted pipeline:

```
stage("Dummy Execution") {
  echo "I am a dummy execution"
}
```

Steps

In steps, we define a particular single task that needs to be executed at a particular time. For example, we can execute the shell "**sh**" and "**echo**" commands inside the steps.

Note: Steps are supported in the declarative pipeline only.

Consider this example:

```
steps {
  sh "make world"
  echo "Hello World"
}
```

Parallel

Parallel is an in-built function of a pipeline that we use to run nested stages parallelly. As we know, Jenkins stages get executed sequentially, and they are dependent on each other. But in some cases, you may need to run the stages parallel to reduce the execution time of Jenkins 'Job'. In such cases,

we can use similar functions, for example: in continuous integration, we can use parallel execution for unit testing and code coverage stages.

The parallel block is supported in both types of the pipeline, i.e., scripted and declarative:

```
parallel {
  stage (“Unit testing”) {
    echo “Unit Testing Execution”
  }
  stage(“Code Coverage”) {
    echo “Code Coverage Execution”
  }
}
```

Shared Library

When we write software or an application, we usually end up writing fewer lines of code ourselves compared to the lines written in pre-existing libraries and dependencies that we use as per our use case. It is possible because someone wrote reusable code before us, which we took advantage of. Let’s take an example; suppose we have to create an application that connects to a MySQL database. Here, we have two approaches: one would be to write a piece of logical code that can connect to MySQL, and another would be to use an existing library for connecting to MySQL.

Now it’s a simple example, but in a real scenario, we cannot write every logic by ourselves, and that will not be good practice.

That’s why we have a library concept that provides us with the functionality of reusable code. The same concept is also present in Jenkins, through which we can develop our own libraries, called “Shared libraries.”

Let’s take a real-world example: suppose we have five Java-based applications, and the CI process for all these applications is similar. Now, if a new service is getting added, we have to duplicate the pipeline again for the sixth application.

Now we can create a common “Shared Library” for Java CI. We can update and modify the shared library for future changes, and the changes will be applied to all the pipelines.

Also, the line of code will get decreased in “**Jenkinsfile**”:

```
@Library (‘opstree-library@master’) _
opstreePipeline()
```

The structure of shared library is as follows:

```
opstree-library
├── resources       #Static files which can be used by shared
library
│   └── input.yaml
├── src             # Sources file for groovy classpath
│   └── org
│       └── opstree
│           └── JavaCIPipeline.groovy
└── vars            # Vars file for src libraries
    └── JavaCIPipelineVars.groovy
```

Examples

There are multiple examples and places where Jenkins pipeline can be used; some of them are given here:

- CI/CD
- Workflow management
- Infrastructure management

CI/CD

CI/CD is one of the major consumers of Jenkins pipeline. We can have detailed visualization of each CI step using the pipeline. Also, we can control what steps need to be run sequentially and what steps need to run parallelly.

In CI/CD, we can leverage the pause functionality of the pipeline as well. For example, we can put a manual approval step before the application deployment. We can also change the slave's node for stages if there is any special requirement.

In brief, we will have great control over our CI/CD pipeline:

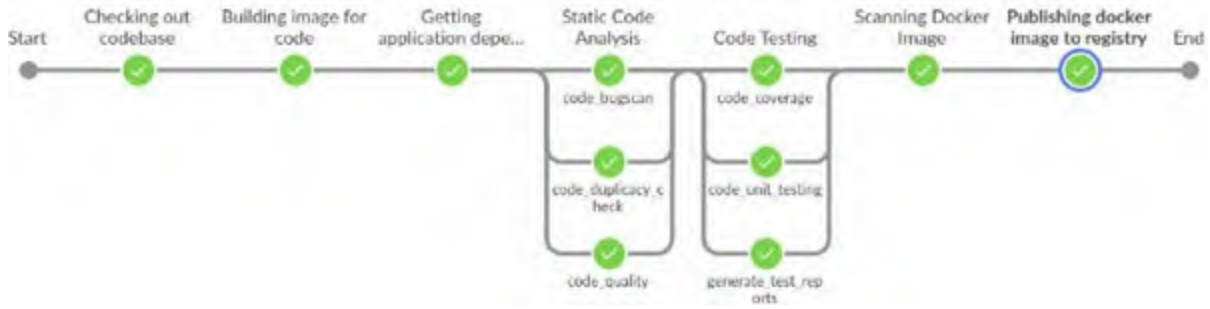

Figure 3.29: *CI/CD pipeline*

Workflow Management

Jenkins pipelines are not restricted to CI/CD; they can be used to manage different types of workflows as well. It means we can integrate a number of tools with pipelines. Suppose we have to design a system that will configure software and packages on remote systems. We can use different tools like ansible or salt-stack. The only issue is that they don't provide the observability and reporting, so we cannot track the changes through them. However, using Jenkins, we can have the observability around the process. Also, we can create a proper visualization of the flow using pipeline:

Figure 3.30: *Workflow management pipeline view*

Infrastructure Management

Jenkins pipelines are widely used for managing the workflow for infrastructure automation as well. Just like we discussed, we like to keep everything in the form of code. The same thing applies to infrastructure as well. We can use Jenkins to keep track of changes in the infrastructure. This doesn't mean that Jenkins will provide the infrastructure for you, but it can be integrated with the tool that is doing the infrastructure provisioning.

Also, infrastructure is an important part of the architecture, and we don't want to mess it up. So, we can have a group of authorized people who can make the infrastructure changes. And the infrastructure creation visualization will be the cherry on the cake.

The flow of infrastructure creation using Jenkins is shown in the following figure:

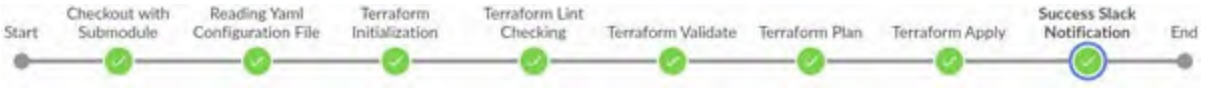

Figure 3.31: *Infrastructure Management pipeline view*

Fact: There are 1000+ open-source shared libraries present on GitHub.

What If the Server Gets Deleted?

The joy of working on the terminal is out of this world, but this delight can turn into misery just by execution of a deletion command. Human beings are prone to make errors, and no one is foolproof. In my opinion, there are two ways of handling it. The first would be to try to make fewer mistakes by being meticulous, but it'll cost us time. The second, which is my preferred way, would be to create a system that can handle our mistakes.

As our infrastructure grows in terms of applications, integrations, users, and pipelines, the Jenkins server will continue to evolve as the backbone of our automation system that we just can't afford to lose.

Backup Plugin Installation

Follow these steps to get started:

1. Install the plugin.
2. Go to **`Manage Jenkins`** | **`Manage Plugins`**.
3. Click on the **`Available`** tab and search for **`Thin backup`**, as shown in the following figure:

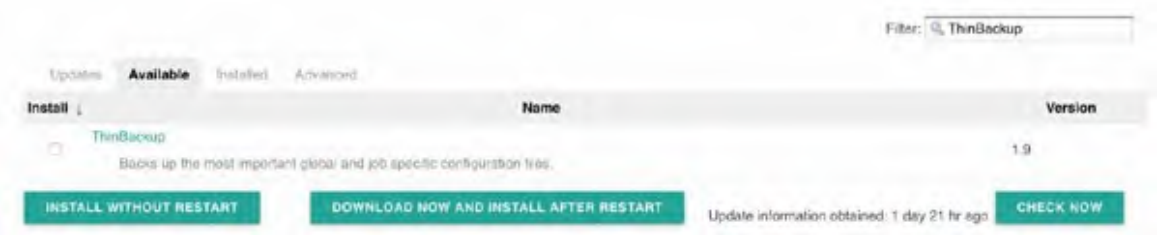

Figure 3.32: *ThinBackup plugin*

4. Install the plugin and restart Jenkins.

Backup Configuration

Once installed, follow these steps for configuring backup settings.

1. Go to **`Manage Jenkins`** | **`ThinBackup`**
2. Click on the **`settings`** option.
3. The **`ThinBackup`** overview is shown in the following figure:

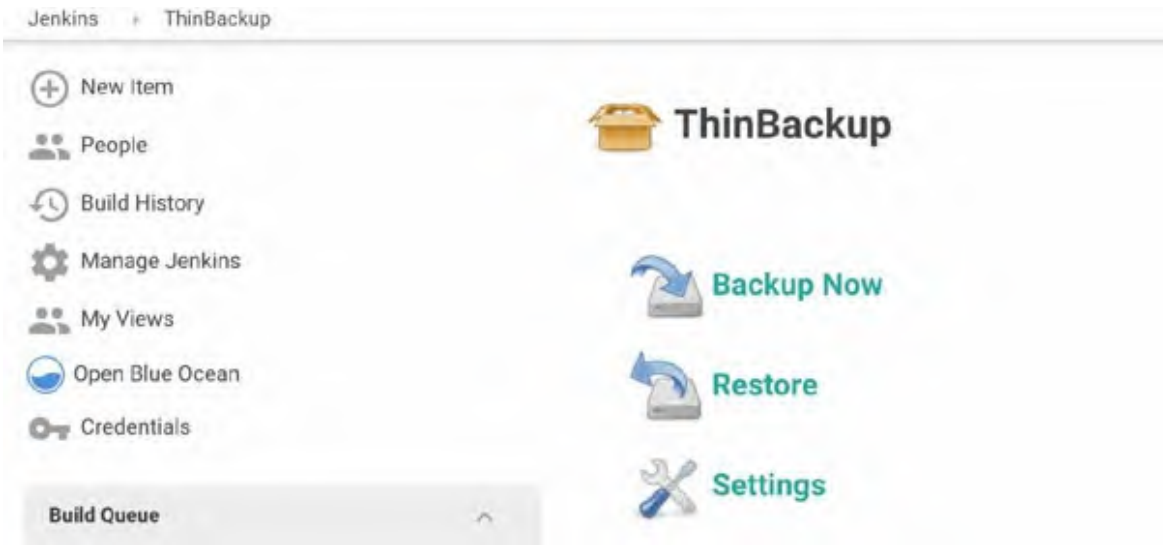

Figure 3.33: *ThinBackup overview*

4. Enter the backup options as shown in the following figure and save them. The backup directory specified should be writable by the user running the Jenkins service. The Jenkins backup will be saved to the backup specified directory, as shown in the following figure:

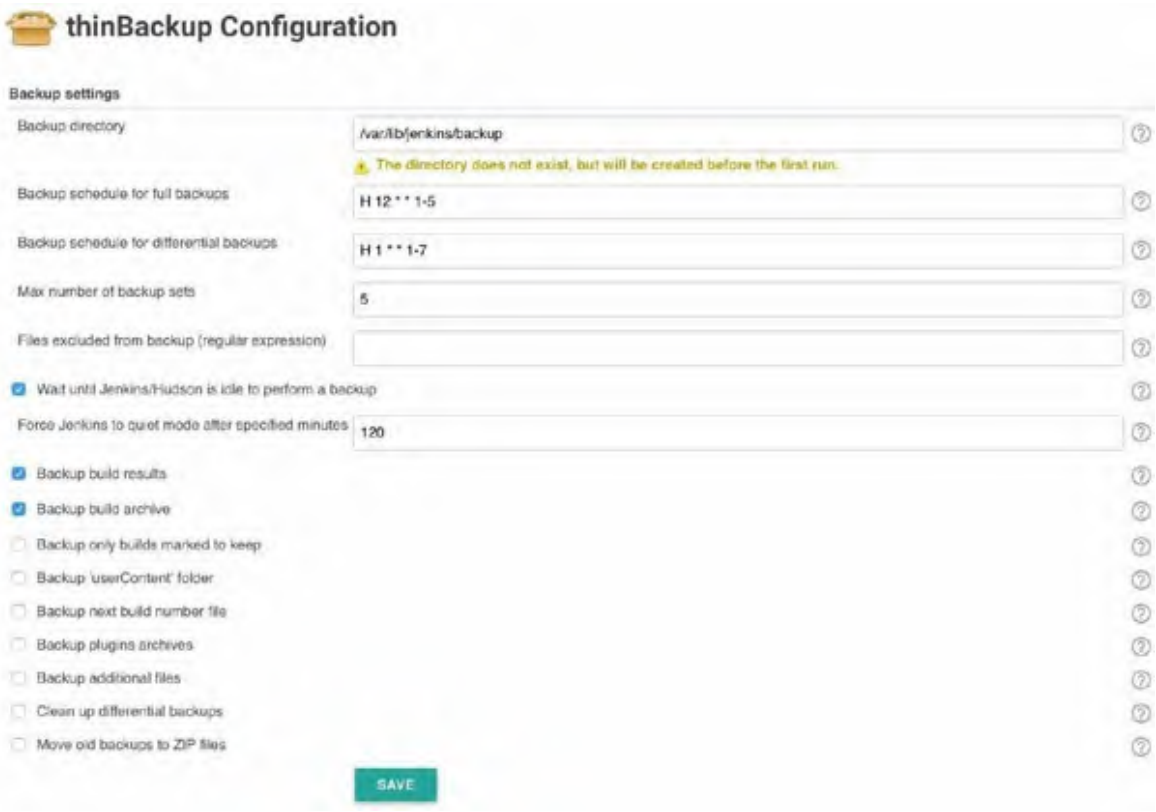

Figure 3.34: *ThisBackup Configuration*

5. Now, click on the **Backup Now** option. It will create a backup of Jenkins data in the specified backup directory in the settings. It is illustrated in the following figure:

Figure 3.35: *Backup Now*

Restoration

The restoration of the backup is even simpler as it lists down the available backups. All we need to do is select and proceed. This will restore the Jenkins configuration to the time when the respective backup was created. Refer to the following figure:

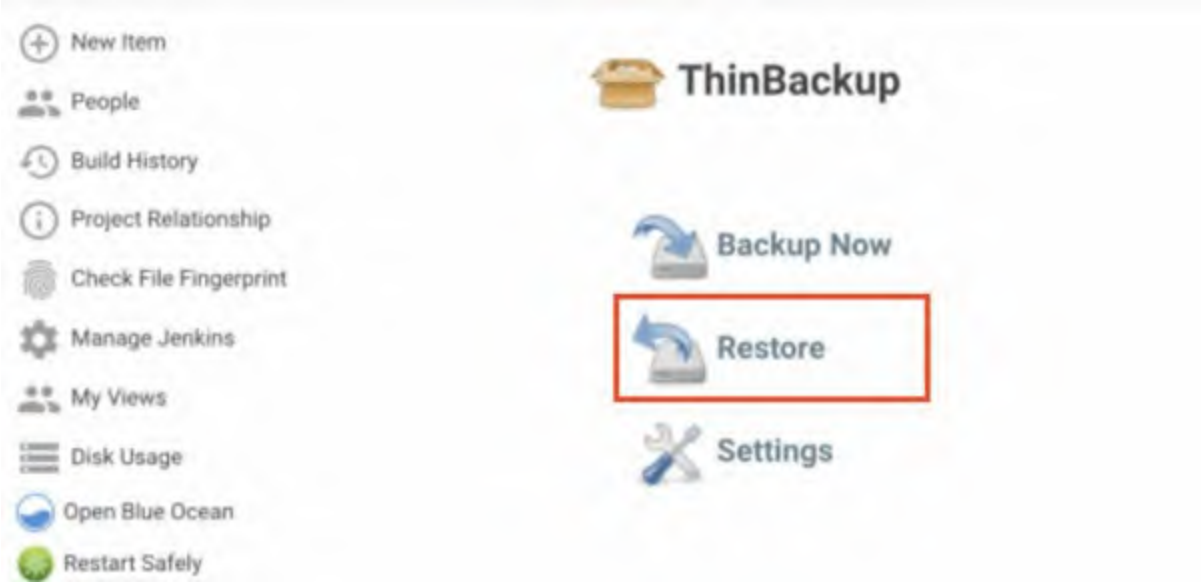

Figure 3.36: *Restore*

Second line of safety (Data Directory Backup and Restore)

During the Jenkins implementation, the best thing I found was that all the settings, build logs, artifact archives, and so on are stored under the **`JENKINS_HOME`** directory. We can simply archive this directory to make a backup. Similarly, restoring the data is just replacing the contents of the **`JENKINS_HOME`** directory from a backup.

```
JENKINS_HOME
 └─ config.xml     (jenkins root configuration)
 └─ *.xml          (other site-wide configuration files)
 └─ userContent    (files in this directory will be served under your http://server/userContent/)
 └─ fingerprints   (stores fingerprint records)
 └─ plugins        (stores plugins)
 └─ workspace (working directory for the version control system)
     └─ [JOBNAME] (sub directory for each job)
 └─ jobs
     └─ [JOBNAME]      (sub directory for each job)
         └─ config.xml     (job configuration file)
         └─ latest         (symbolic link to the last successful build)
         └─ builds
             └─ [BUILD_ID]     (for each build)
                 └─ build.xml      (build result summary)
                 └─ log            (log file)
                 └─ changelog.xml  (change log)
```

Figure 3.37: *JENKINS_HOME directory*

Third line of safety (Jenkins Server Image)

I still remember the days when I created images of the whole system on a disk and called it a Clone Disk. That had been a lifesaver for me several times. I am not surprised to say that we can use the same strategy to back up and restore the Jenkins server as well.

Be it a self-hosted infrastructure or cloud managed, both provisioners use the capability of creating an image of the server and restoring it whenever needed.

Fact: Jenkins crash without backup costs us a complete day and night to make it live again.

Master/Slave architecture

If you have watched the “Batman” trilogy, you would remember Batman having a butler or Jenkins named “Alfred”. But Alfred didn’t do all the work himself; he had the support of other butlers who were working under him.

The similarities between Alfred and Jenkins servers are noteworthy, they can do a lot of operations, but they both need helping hands when they have to do parallel processing.

In Jenkins, we have a concept of slave that helps Jenkins perform many tasks parallelly.

Now, the question is, ‘If we can scale Jenkins hardware or configurations as well, then why do we need the slave’s concept?’. We agree that we can scale its hardware or configurations, but there could be some drawbacks to that, like Jenkins downtime, Single OS support, and OS corruption. To overcome such issues, Jenkins uses a master-slave architecture to manage distributed builds. The main role of the Jenkins master is to schedule and dispatch builds for execution on slave or master depending upon the job configuration. On the other hand, a Jenkins slave is a system that takes orders from the master and executes tasks according to them.

Configuring master-slave in Jenkins is no rocket science. We can set up this distributed architecture easily. Also, Jenkins supports different types of slaves that we can leverage as per our requirements:

- JNLP slave

- Dynamic slaves
- SSH-based slaves

JNLP Slaves

The JNLP slaves are commonly used with dynamic slaves, but it doesn't mean that we cannot use them with static servers. JNLP stands for Java Network Launch Protocol) and is used to connect remote systems to Java.

In order to add a JNLP slave, we have to enable the JNLP port inside Jenkins. We can enable the JNLP port in **`Manage Jenkins`** | **`Configure Global Security`** | **(Agents Section)**.

This is illustrated in the following figure:

Figure 3.38: *JNLP Slaves - agent config*

Once the JNLP port is enabled, we have to configure the slave. First, we have to configure the master to use the JNLP slave.

`Manage Jenkins > Nodes`

Refer to the following figure:

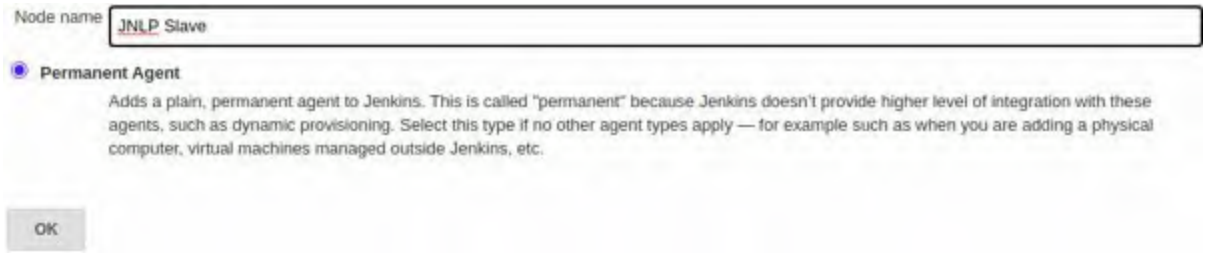

Figure 3.39: *JNLP Slaves – Node config*

Once the slave information is added to the master, we have to download the JNLP jar file from the master.

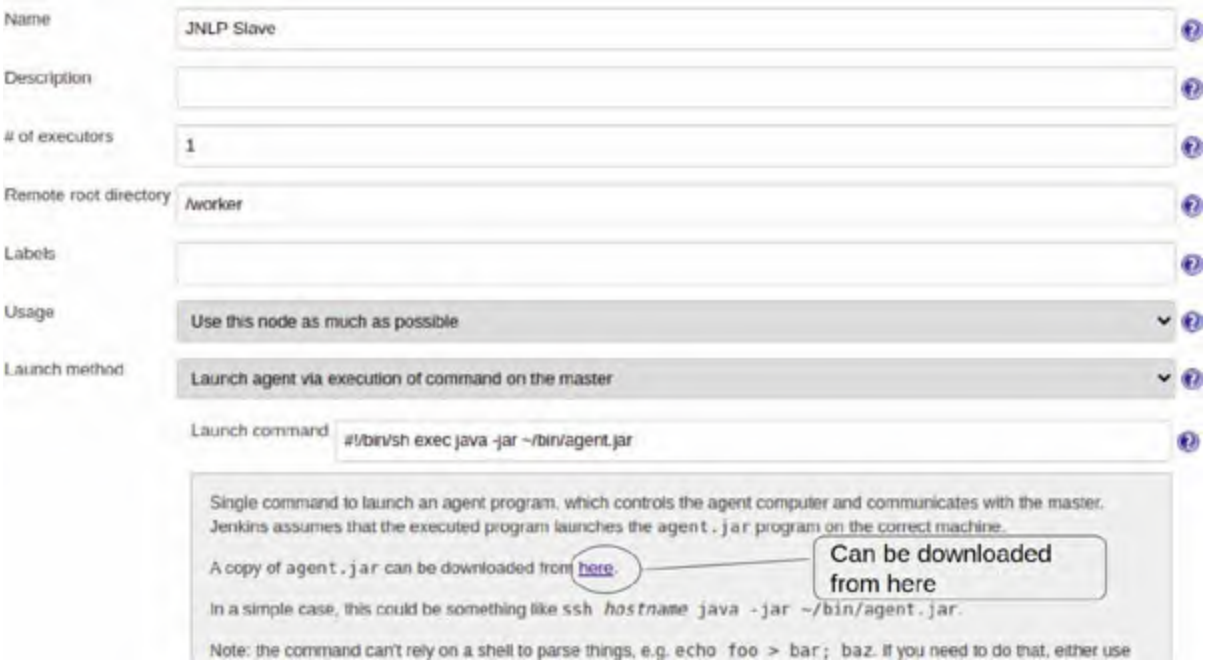

Figure 3.40: *Download the JNLP jar file from the master*

On the slave system, we must execute this command:

```
$ java -jar ~/bin/agent.jar
```

After this, the slave will be added successfully and can be used in any job.

SSH Slaves

SSH slaves also use the JNLP behind the scenes. The only difference is that we don't have to configure the JNLP slave by ourselves and simply have to provide the SSH connection details of the slave.

We can simply add the SSH-based slave by going to **Manage Jenkins | Nodes.** Then, we must provide the connection details of the slave server. Consider this example:

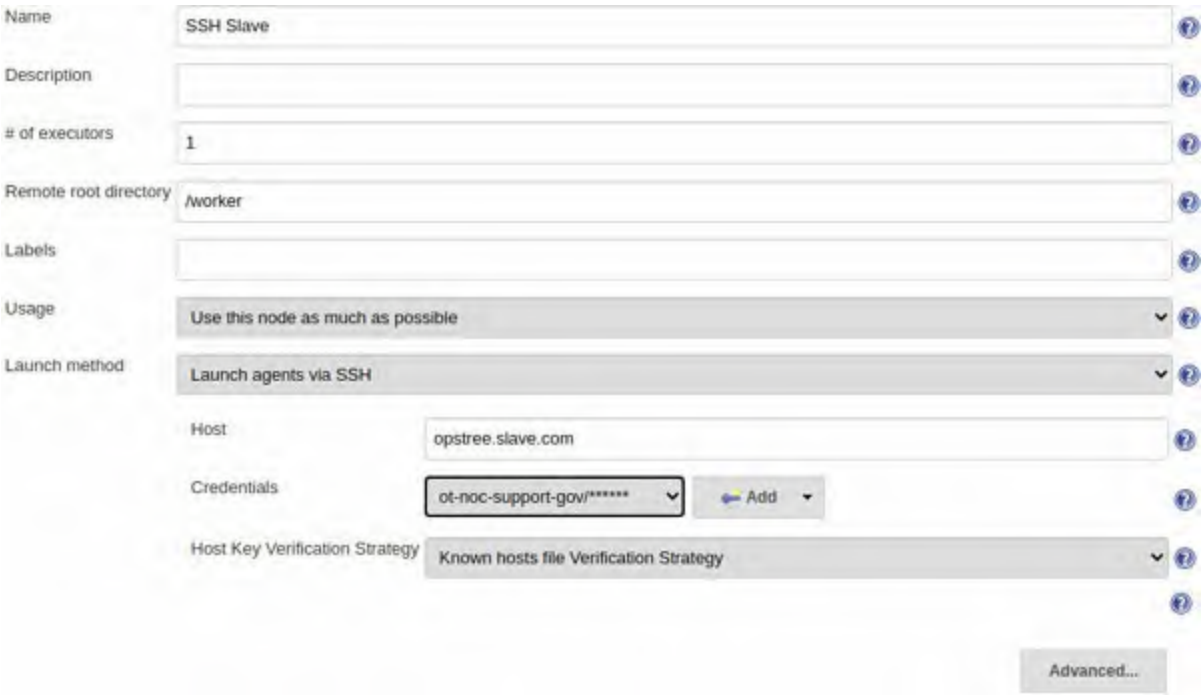

Figure 3.41: *SSH slave configs*

Once we save the configuration, we can check the logs for validation for whether the slave is successfully added.

Figure 3.42: *Verify agent(slave) connectivity*

Dynamic Slaves

Some organizations worship cost-cutting methods, and the dynamic slaves method can be handy in those scenarios. Let’s assume that we have to execute a few sets of tasks daily around 03:00 PM and have created a slave with decent resources to execute these tasks. However, the resource of that slave will only be used at 03:00 PM. Apart from that, the resource will be a waste to us.

For handling scenarios like this, we can use the dynamic slave’s concept of Jenkins in which our slave will be spawned on-demand and after the execution, it will be automatically terminated.

The dynamic slaves could be of any type like- Docker container, Kubernetes pods, or Cloud instances like AWS EC2.

We can configure the dynamic slave from **`Manage Jenkins`** | **`Nodes`** | **`Cloud`** | **`Add Cloud.`**

Refer to the following figure:

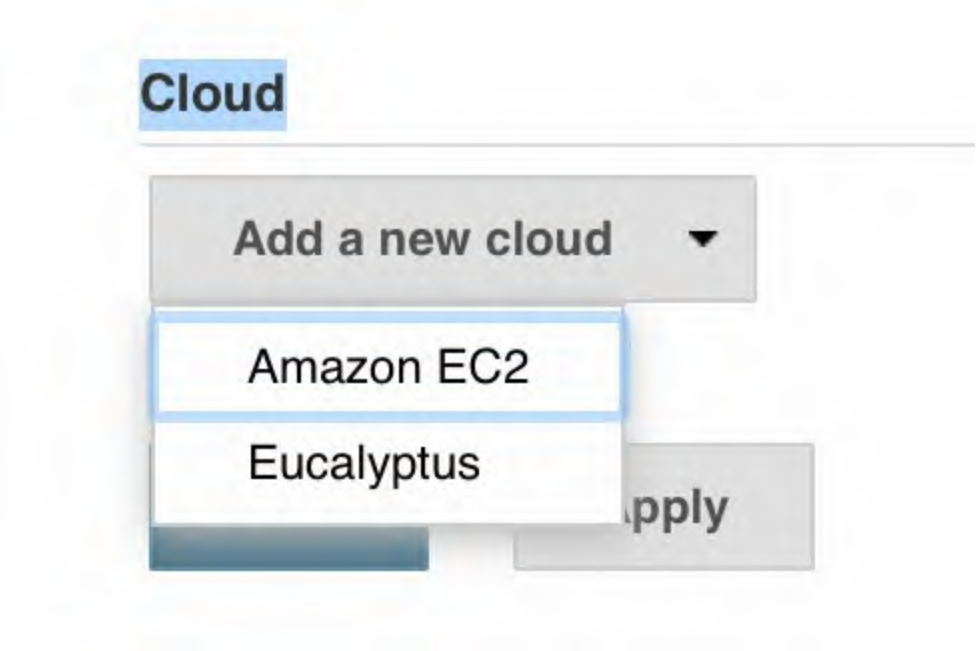

Figure 3.43: *Dynamic Slaves*

Scenarios

There are multiple scenarios where Jenkins slaves can be a great helping hand for us. For example, suppose we are building an application that supports multiple OS platforms; in that case, we can add different types of OS like Windows, and macOS as Jenkins slaves and compile the application on them.

Also, it will reduce the package installation burden from the master node because if any package got broken or corrupted, it can corrupt the complete Jenkins server, and we might have to redo everything from scratch.

We can use slaves with the integration of emulators as well to build different application projects like Android or IOS apps. Since mobile application testing requires a decent amount of resources, we cannot do the complete testing on the master itself as we need to over-provision it. But if we use Jenkins slave in this type of testing, we will have control over bringing slaves

online and offline as per our need. This will help us in saving the infrastructure cost as well.

Fact: There are 20+ plugins in Jenkins to manage slaves in Cloud like AWS, GCP, Openstack, and Kubernetes.

Global tool configuration

During the course of the Jenkins Server setup there are multiple configurations came across. One of them is the Global Tool Configuration that will come in handy while creating the Jenkins Job. It contains configurations related to JDK, Maven, Ant, Gradle, Docker, Sonarqube, and others.

Figure 3.44: *Global tool configuration*

Jenkins provides support for working with multiple JDK versions. The simplest way to declare a JDK installation is to simply supply an appropriate name along with the path to the Java installation directory. Refer to the following figure:

Figure 3.45: *Configure JDK*

The project will be following a standard, well-defined build life cycle of compile, test, package, deploy. Each life cycle phase is associated with a Maven plugin. Jenkins provides excellent support for Maven and has a good understanding of Maven project structures and dependencies.

We can either get Jenkins to install a specific version of Maven automatically or provide a path to a local Maven installation. The cherry on top is its capability to support different versions of Maven for different projects.

In *figure 3.46*, the Maven installation using Jenkins is shown:

Figure 3.46: *Configure Maven*

We can have other tools installed the same way as JDK and Maven:

SonarQube Scanner installation:

Figure 3.47: *Configure SonarQube Scanner*

Synk installation is shown in the following figure:

Figure 3.48: *Configure Synk*

Docker installation is shown in the following figure:

Figure 3.49: *Configure Docker*

Conclusion

With all this information, convincing everyone was easy. In the latest CI walkthrough meeting, I went all out with my presentation of various topics.

There were a few questions regarding the nuances of the process. Considering the number of people involved, I expected at least this much. It wasn't anything I didn't prepare for. Thankfully, everyone was positive about this change in the infrastructure, and I got a green ticket to implement CI. What more could one ask for? I pulled up my socks as now was the time for action.

In the next chapter, we will focus on CI pipelines. We will learn to create them and add multiple stages using declarative syntax to perform complete CI.

Points to Remember

- Jenkins is not only a CI tool; it's an automation tool that can work as a Swiss knife.
- Jenkins is not restricted by functionality; we can easily extend the Jenkins functionalities by adding or creating plugins.
- Jenkins has a scalable nature; we can add slave nodes to Jenkins to manage multiple jobs.

Multiple choice questions

1. **How can we add slaves to the Jenkins server?**
    a. JNLP Slaves
    b. SSH Slaves
    c. Dynamic Slave (Docker/Kubernetes)
    d. All of the above
2. **Which is not a pipeline terminology?**
    a. Node
    b. Steps
    c. Stage
    d. Slave

3. **Can Jenkins jobs be maintained as pipeline as code?**
    a. True
    b. False

Answers

1. d
2. d
3. a

Questions

a. How can we take backup in Jenkins, and what's the efficient way to take Jenkins backup?
b. What is the configuration change we have to make to restrict Jenkins's concurrent process?
c. What's the difference between authentication and authorization?
d. What is pipeline as code inside Jenkins?

Key terms

- Pipelines
- Plugins
- Global Tool Configuration
- JNLP Slave
- Dynamic Slave

CHAPTER 4

CI with Jenkins

It was time for action. Since all the CI points were clear in my head and the tool was finalized, the next thing was to start implementing the CI for this project. The rule for implementation is to test everything on a dummy project and perform a detailed POC before stepping into production. This is why we have multiple environments to test every aspect of our new project.

Structure

The following topics will be discussed in the chapter:

- CI Pipeline with Pre-Deployment Steps integration with Jenkins
    - Code Stability
    - Code Quality
    - Unit Testing
    - Security Testing
    - Sonarqube Integration
- CI Pipeline with Intermediate Steps integration with Jenkins
    - Artifacts Management
    - Dev Environment Deployment
    - DB Update
- Notifications with Jenkins

Objectives

After going through this chapter, you should understand how different tools can be used to execute **Continuous Integration** (**CI**) checks. Also, you should be able to integrate different CI steps like Code Quality, Stability, and

Unit Testing in the Jenkins pipeline. This chapter will also discuss the importance of notification strategy in real-life scenarios and how we can integrate notifications into our Continuous Integration pipelines.

CI Pipeline with Pre-Deployment Integration Checks

Before jumping to Jenkins, I performed all the steps on a development (dev) machine. Hence, I quickly found a project available on VCS and started my experiments.

The project is called "**`Spring3Hibernate`**," and is written in Java. It is available at the preceding location. I went through the project README, found prerequisites and quickly resolved those:

- Maven 3.X
- Jdk 8.

Once it was done, I started compiling the project to check the stability of the code. Since I was using Maven as a build tool, I had to execute goals for my tasks. For code stability, the command is as follows:

```
$ mvn clean package
```

Once the order successfully is executed, we get this as output:

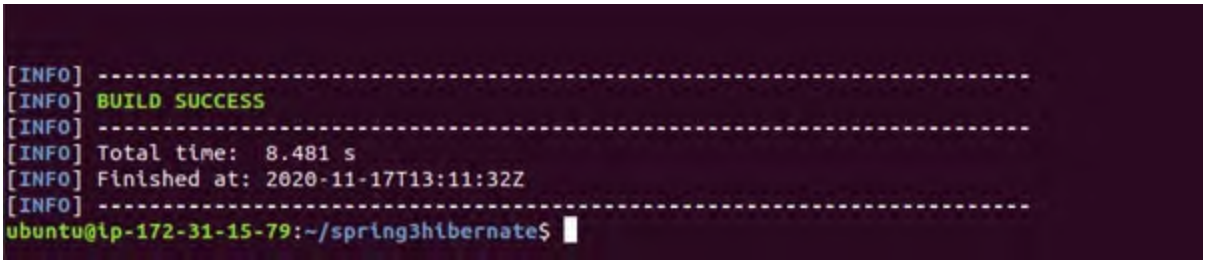

Figure 4.1: *Maven command output for compilation*

Like code stability, I was using the "**`Checkstyle`**" plugin for code quality as well. Checkstyle is a plugin used by maven to maintain code best practices and quality in Java-based projects, which we discussed earlier. The goal execution is as follows:

```
$ mvn checkstyle:checkstyle
```

The output of the "**`Checkstyle`**" command is shown in the following figure:

Figure 4.2: *Maven command output for checkstyle*

For more information about the check style, refer to their official website at the following link:

https://checkstyle.sourceforge.io/

Well, the developers of the project have written some test cases for the application, so it doesn't take a lot of effort to execute the unit testing on the application. Next, we simply need to run the following command:

```
$ mvn test
```

All the unit tests are available inside the **src/test/java/com** directory:

https://github.com/bpbpublications/CI-CD-Simplified/tree/main/src/test/java/com/sample

Figure 4.3 illustrates the console output of unit test case:

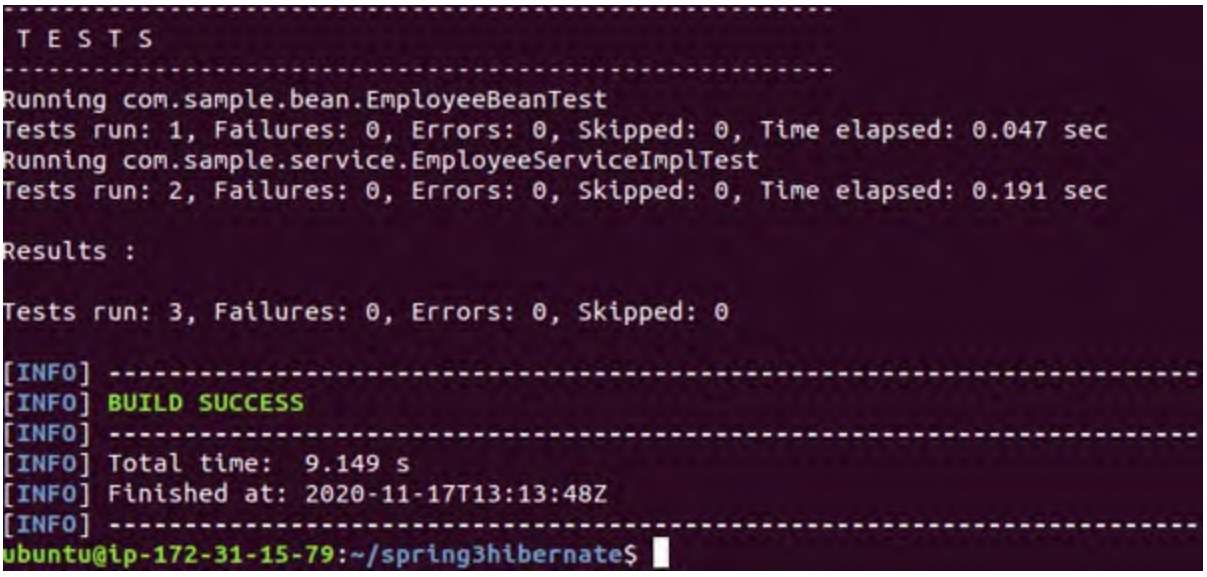

Figure 4.3: *Maven command output for unit test execution*

Having completed Code Stability, Code Quality, and Unit Testing easily, it's time for Code Coverage. Coverage metrics require using another maven plugin called "**`Cobertura`**." We will execute the goal as follows:

```
$ mvn cobertura:cobertura
```

For more information about the Cobertura, go through the official documentation at the following link:

https://cobertura.github.io/cobertura/

The output of the Cobertura command is shown here:

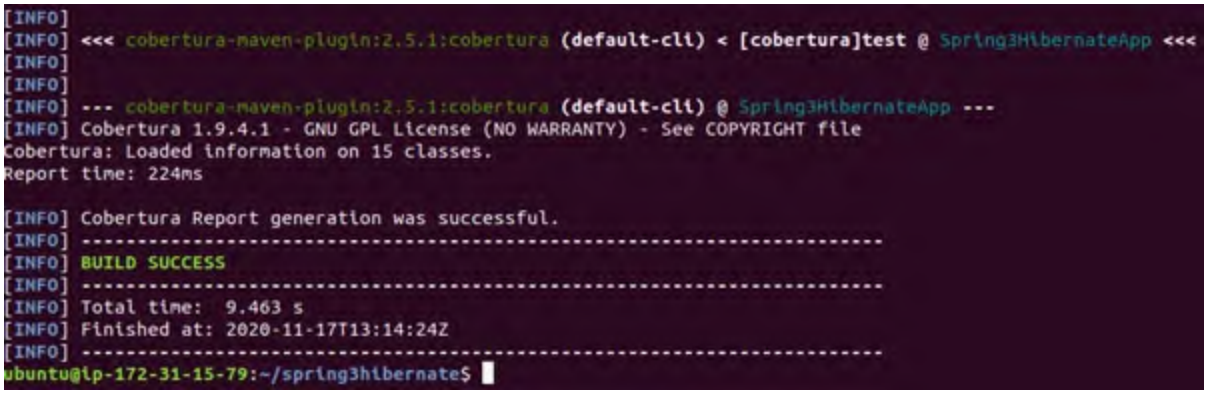

Figure 4.4: *Maven command output for Cobertura*

As for security testing, one needs to scan the dependencies of the project for vulnerabilities. To achieve this, let's use a popular "OWASP" dependency vulnerability scanning and add a goal in the maven configuration for the same:

```
$ mvn org.owasp:dependency-check-maven:check
```

For more information about the OWASP project, go through the official documentation at the following link:

https://owasp.org/

Last but not least, we need to publish all the different reports to a single platform, where it would be convenient to analyze the trend and take the necessary action. It was time to integrate the famous SonarQube with our maven setup. To publish the report on sonar, we simply have to run the following command:

```
$ mvn sonar:sonar -Dsonar.host.url=${SONAR_URL} -
Dsonar.login=${SONAR_USER} -Dsonar.password=${SONAR_PASSWORD} -
Dsonar.java.binaries=.
```

In the preceding command, Sonarqube will start analyzing the project for different parameters, like the code's quality, security threats, and maintainability of code. Once the analysis is complete, it will upload the report on Sonarqube, and it can be accessed on its URL.

http://<sonar_qube_url>:<sonarqube_port>/

Once we execute these steps in dev, it is time to automate them using dev Jenkins. So, let's log in to the Jenkins server, and create a Folder named "CI" and a pipeline project named "**`Spring3Hibernate`**", as shown in the following figure:

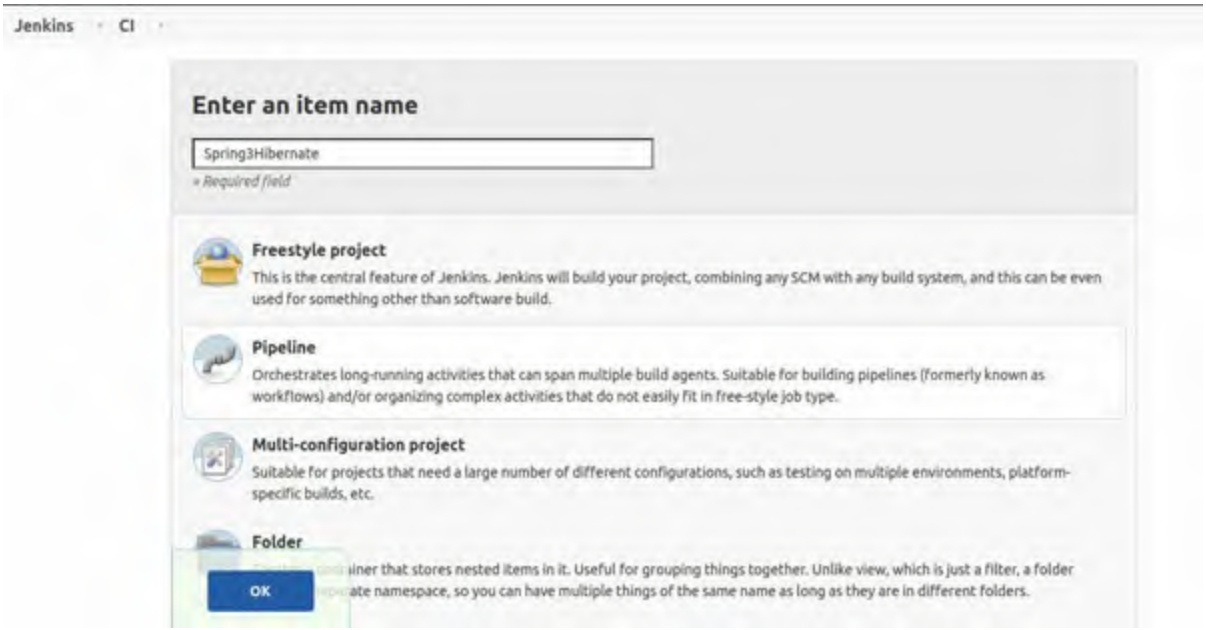

Figure 4.5: *Jenkins job creation options*

Note: Many important plugins need to be installed in Jenkins to make the CI pipeline effective. Maven and jdk8 should be installed on the system as packages. Also, warnings-ng, workflow-scm-setup, pipeline, slack, and email plugins should be installed on the system.

Code Checkout

We have already discussed a Jenkins pipeline and its sequential or parallel workflows in detail. Keeping that as the base, the first step we need to take is that of cloning the repository. There is already a plugin installed in Jenkins with the name **workflow-scm-setup** for this purpose.

For generating the code, we will use "Pipeline Syntax Generator" and fill out the details as shown here:

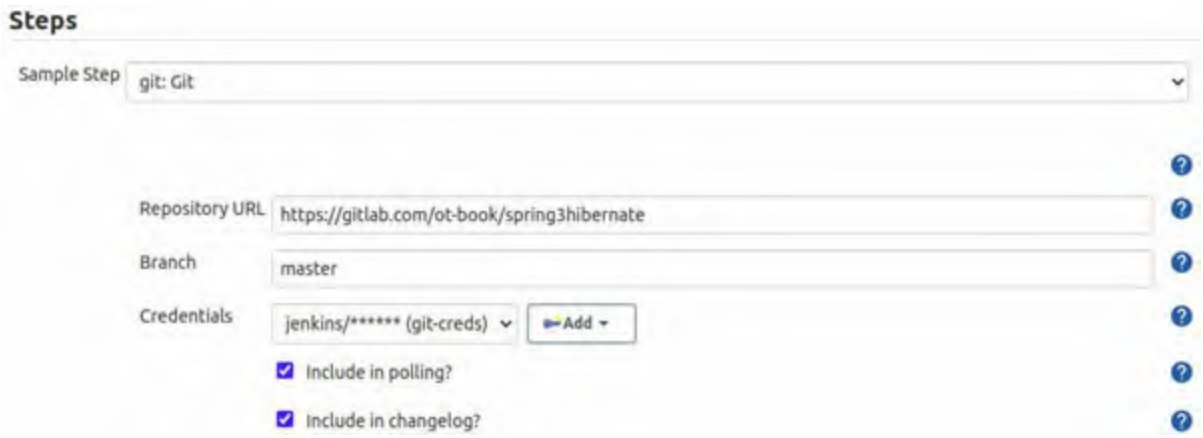

Figure 4.6: *SCM Configuration of Jenkins*

Once the syntax is generated, we just need to copy-paste the code inside the Pipeline Script Section of the Job. The code should look like this:

```
node("master") {
   stage("Cloning Spring3Hibernate Project") {
     git url: 'https://github.com/bpbpublications/CI-CD-
     Simplified'
   }
}
```

And the Pipeline Script Section will be as shown here:

Figure 4.7: *Pipeline script for Code Checkout*

Now, merely by saving the job and executing it with the "**`Build Now`**" button, we could see stage execution in the "**`Stage View`**" of the pipeline:

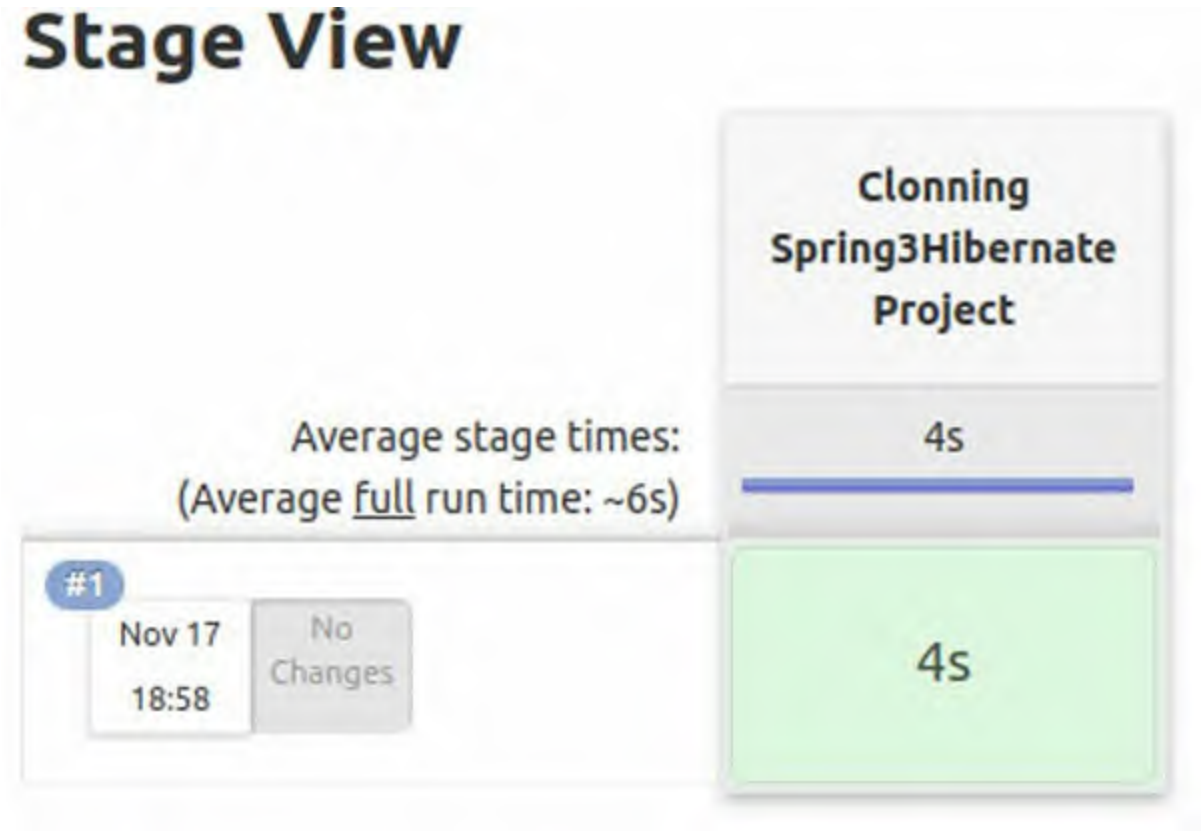

Figure 4.8: *Stage view of code checkout*

Also, if we click on the build number, we will be able to see the console logs of the job.

Code Stability

Once the repository is cloned, we will write the second stage for checking the code stability, as follows:

```
node("master") {
   stage("Cloning Spring3Hibernate Project") {
     git url: 'https://github.com/bpbpublications/CI-CD-
     Simplified'
   }
   stage("Code Stability") {
     sh "mvn clean install"
   }
}
```

Following is the "**`Stage View`**" with the second stage:

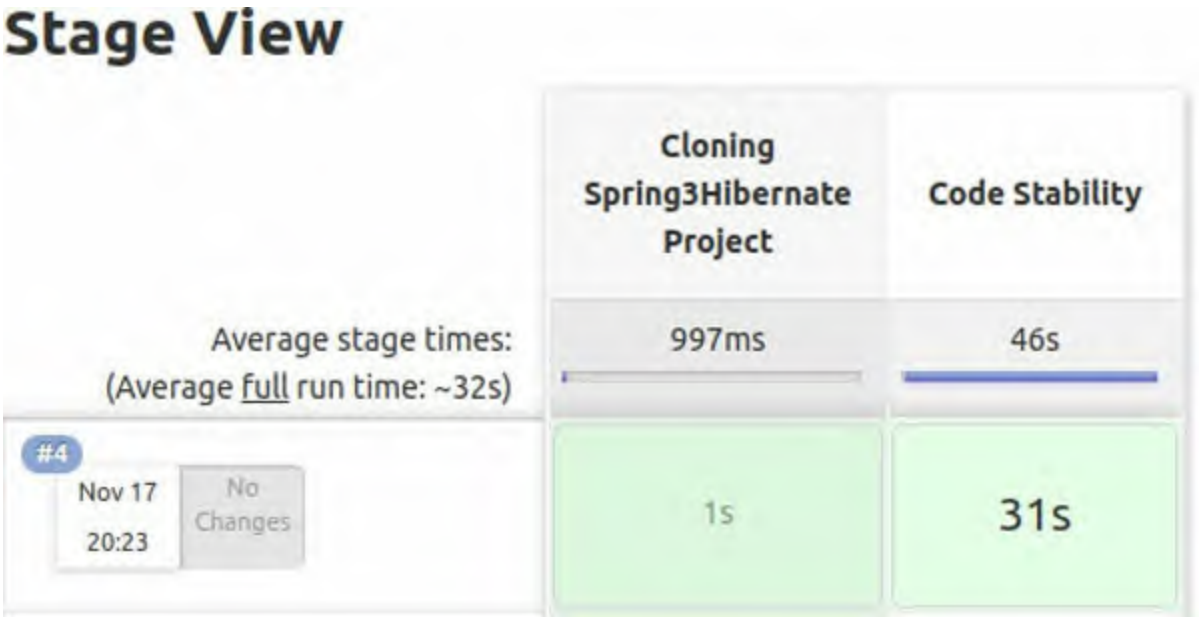

Figure 4.9: *Stage view with code stability*

Code Quality

Now is the time to add our third stage for "*Code Quality*." Here, we will look out for a report as well. Therefore, we will use the Jenkins warnings-ng plugin for report publishing:

```
node("master") {
   stage("Cloning Spring3Hibernate Project") {
     git url: 'https://github.com/bpbpublications/CI-CD-
     Simplified'
   }
   stage("Code Stability") {
     sh "mvn clean install"
   }
   stage("Code Quality") {
     sh "mvn checkstyle:checkstyle"
     recordIssues(tools: [checkStyle(pattern: '**/checkstyle-
     result.xml')])
   }
}
```

Post execution, we get the following output:

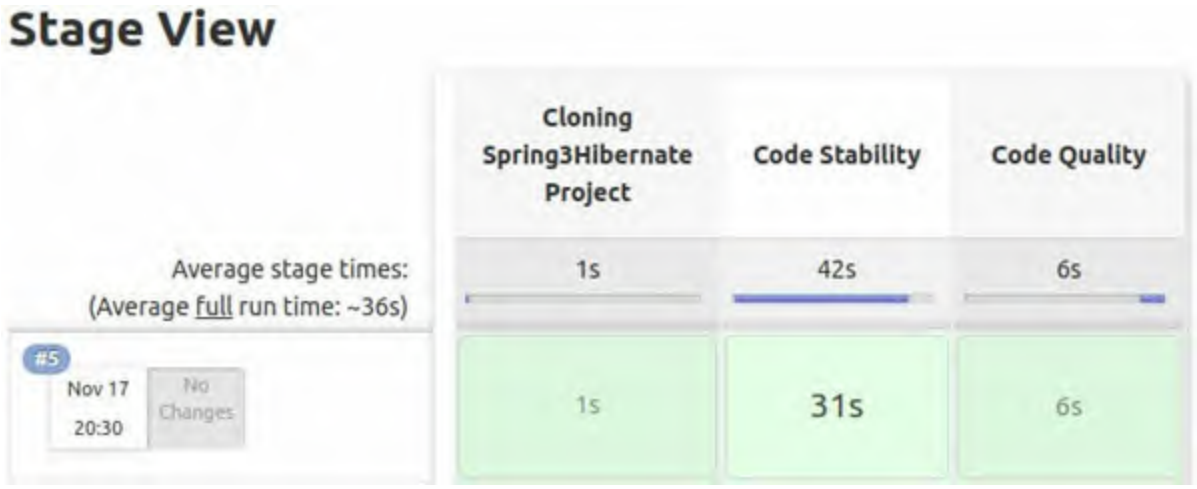

Figure 4.10: *Stage view with code quality*

To access the checkstyle report, go to the Jenkins job console and click on the "**`Checkstyle`**" report. *Figure 4.11* illustrates the Checkstyle report:

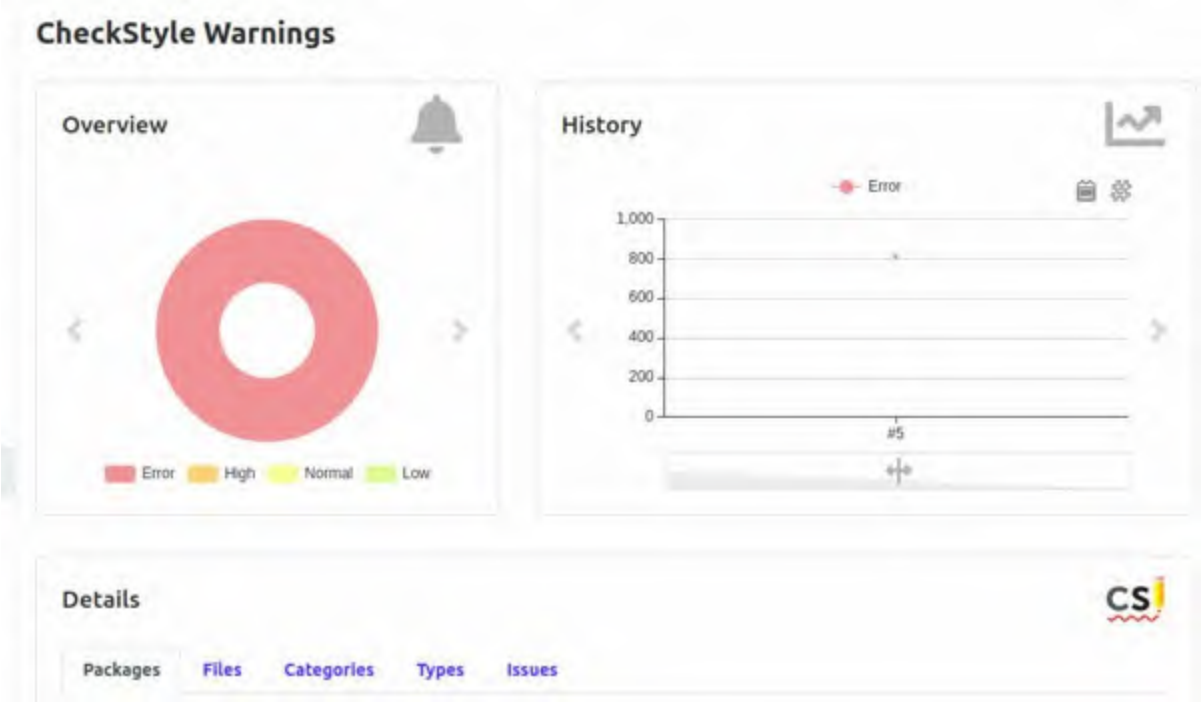

Figure 4.11: *Checkstyle report*

Unit Testing

Since the test cases are already written, we simply need to execute the test cases and publish the report for "*Unit Testing*." Again, we will use the warnings-ng plugin with the JUnit reporting feature to publish the

information:

```
node("master") {
   stage("Cloning Spring3Hibernate Project") {
     git url: 'https://github.com/bpbpublications/CI-CD-
     Simplified'
   }
   stage("Code Stability") {
     sh "mvn clean install"
   }
   stage("Code Quality") {
     sh "mvn checkstyle:checkstyle"
     recordIssues(tools: [checkStyle(pattern: '**/checkstyle-
     result.xml')])
   }
   stage("Unit Testing") {
     sh "mvn test"
     recordIssues(tools: [junitParser(pattern: 'target/surefire-
     reports/*.xml')])
   }
}
```

Once the code is updated, triggering the "Build Now" button again will give us this "Stage View" with Unit testing and a "JUnit Warnings" result, as shown in the following image:

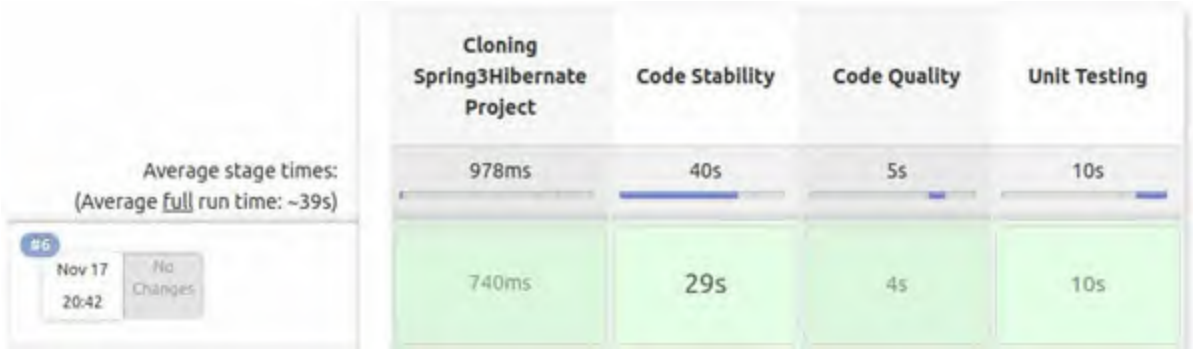

Figure 4.12: *Stage view with Unit Testing*

To access the unit testing report, go to the Jenkins job console and click on "JUnit report". *Figure 4.13* shows the Junit report:

Figure 4.13: *Junit report*

Security Testing

It requires adding another stage to the pipeline and another plugin in Jenkins, called the HTML publisher plugin, for its HTML report. The code for this is as follows:

```
node("master") {
   stage("Cloning Spring3Hibernate Project") {
     git url: 'https://github.com/bpbpublications/CI-CD-
     Simplified'
   }
   stage("Code Stability") {
     sh "mvn clean install"
   }
   stage("Code Quality") {
     sh "mvn checkstyle:checkstyle"
     recordIssues(tools: [checkStyle(pattern: '**/checkstyle-
     result.xml')])
   }
   stage("Unit Testing") {
     sh "mvn test"
     recordIssues(tools: [junitParser(pattern: 'target/surefire-
     reports/*.xml')])
   }
   stage("Security Testing") {
```

```
        sh "mvn org.owasp:dependency-check-maven:check"
        publishHTML([allowMissing: false, alwaysLinkToLastBuild:
        false, keepAll: false, reportDir: 'target', reportFiles:
        'dependency-check-report.html', reportName: 'Dependency
        Check Report', reportTitles: ''])
    }
}
```

After the build is completed with the updated code, we will have a stage view as shown in the following figure:

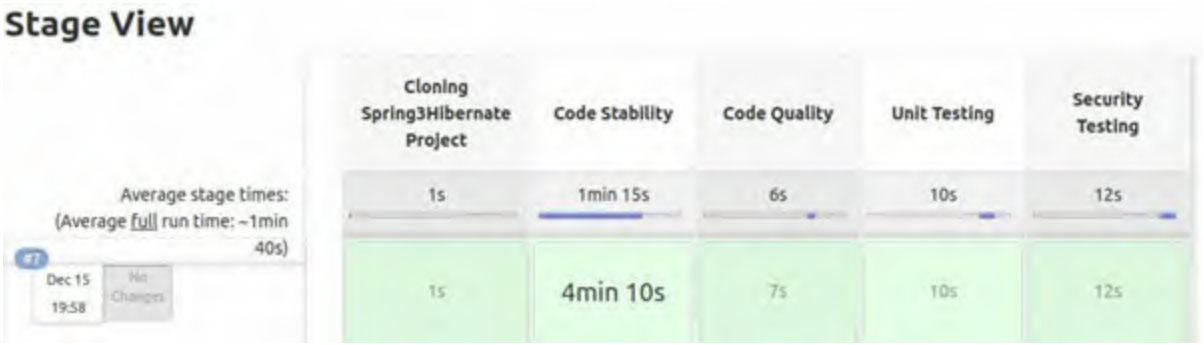

Figure 4.14: *Stage view with Security Testing*

To access the OWASP report, go to the Jenkins job console and click on "**Dependency Check Report**". *Figure 4.15* illustrates the OWASP dependency check report:

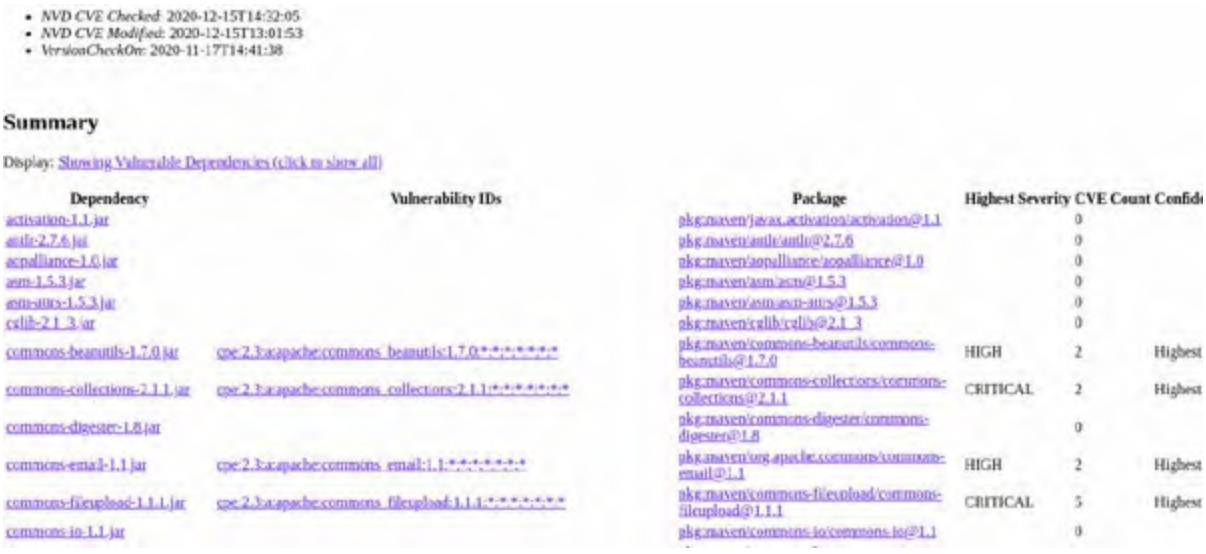

Figure 4.15: *OWASP Dependency check report*

Sonarqube Integration

For integration with Sonarqube, we need to make some changes in the Jenkins Global Configuration so that every project can leverage it.

Sonarqube can be installed independently on a Linux or Windows system. Also, if you don't want to manage Sonarqube to reduce operations, it also provides a cloud offering as well.

For more information about installation and configuration, check out the official documentation at the following link:

https://docs.sonarqube.org/latest/setup/install-server/

Go to **`Manage Jenkins`** > **`Configure System`** and add environment variables, as shown in the following figure:

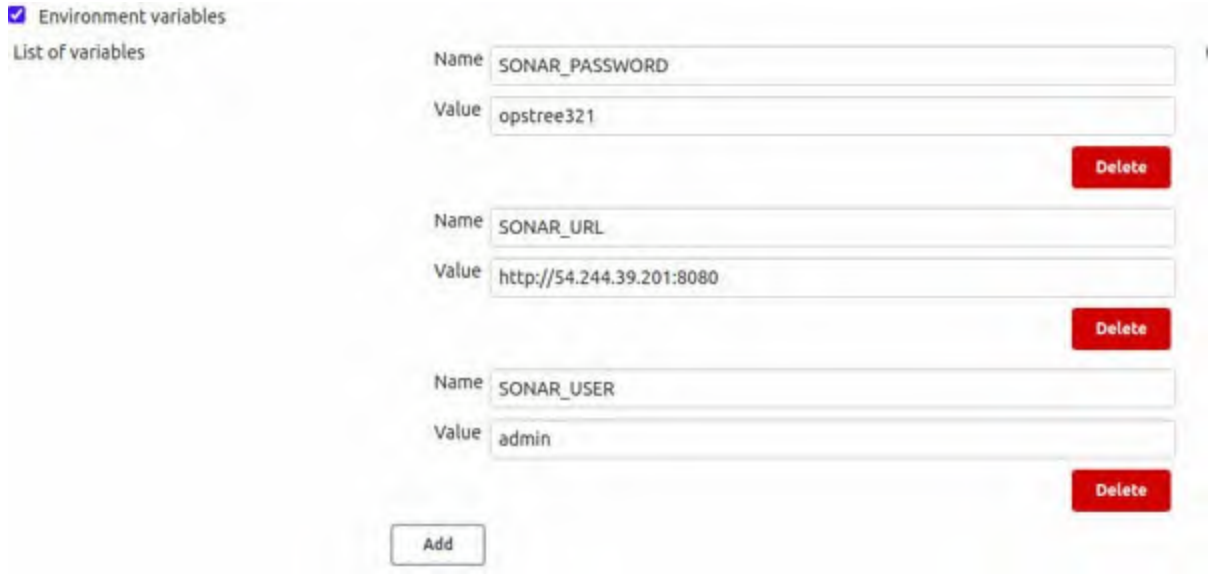

Figure 4.16: *Sonarqube Environment Variables*

Then, we will go ahead to update the pipeline code for Sonarqube integration, as follows:

```
node("master") {
   stage("Cloning Spring3Hibernate Project") {
     git url: 'https://github.com/bpbpublications/CI-CD-
     Simplified'
   }
   stage("Code Stability") {
     sh "mvn clean install"
   }
   stage("Code Quality") {
     sh "mvn checkstyle:checkstyle"
     recordIssues(tools: [checkStyle(pattern: '**/checkstyle-
```

```
    result.xml')])
  }
  stage("Unit Testing") {
    sh "mvn test"
    recordIssues(tools: [junitParser(pattern: 'target/surefire-
    reports/*.xml')])
  }
  stage("Security Testing") {
    sh "mvn org.owasp:dependency-check-maven:check"
    publishHTML([allowMissing: false, alwaysLinkToLastBuild:
    false, keepAll: false, reportDir: 'target', reportFiles:
    'dependency-check-report.html', reportName: 'Dependency
    Check Report', reportTitles: ''])
  }
  stage("Sonarqube Analysis") {
    sh "mvn sonar:sonar -Dsonar.host.url=${SONAR_URL} -
    Dsonar.login=${SONAR_USER} -
    Dsonar.password=${SONAR_PASSWORD} -Dsonar.java.binaries=."
  }
}
```

And the output with SonarQube analysis is shown here:

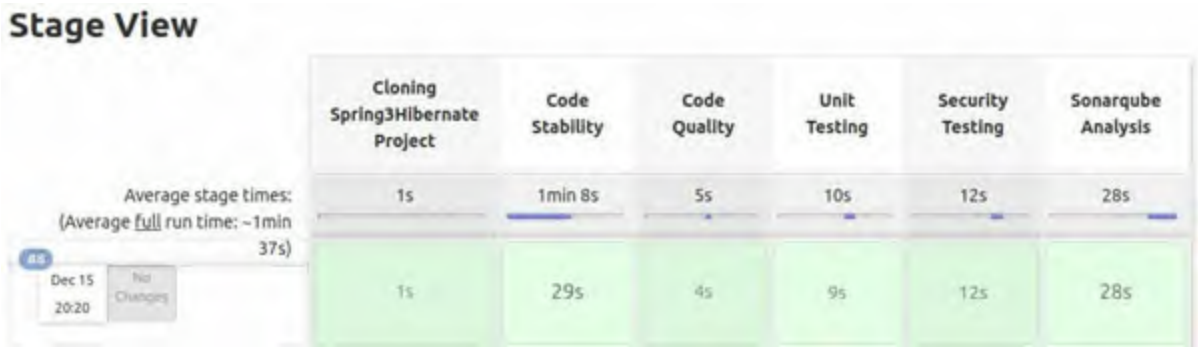

Figure 4.17: *Stage view with Sonarqube Analysis*

Figure 4.18 shows the Sonarqube report:

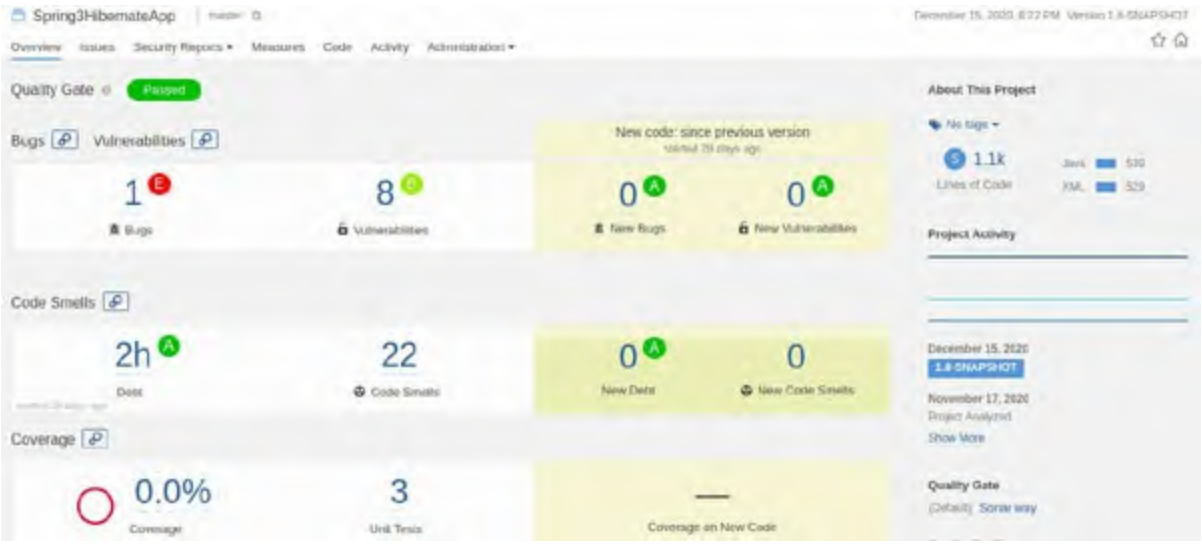

Figure 4.18: *Sonarqube report*

Converting Multibranch Pipeline

The Multibranch pipeline is a Jenkins plugin that helps in creating different pipeline jobs corresponding to the branch in which a Jenkins file is located. So, we will create a Jenkins file in the master branch of the repository and add the pipeline code to it, as shown in the following figure:

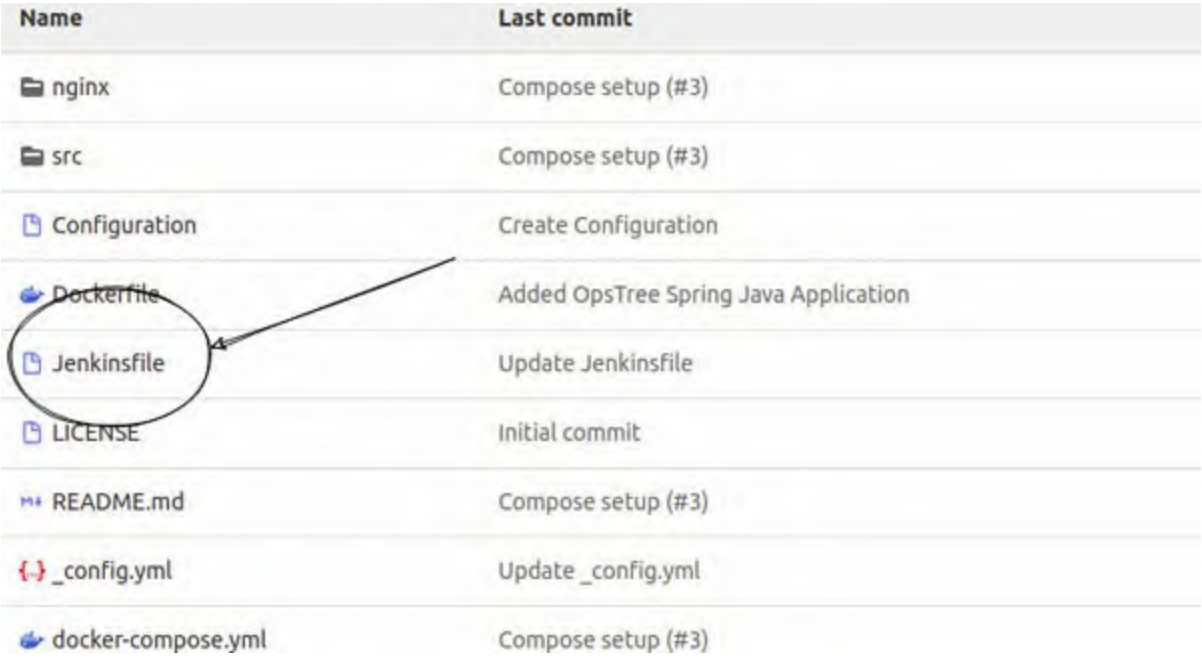

Figure 4.19: *Repository structure*

In Jenkins, we need to create a new job named "**`Spring3hibernate-`**

`Multibranch-CI`", as shown in the following screenshot:

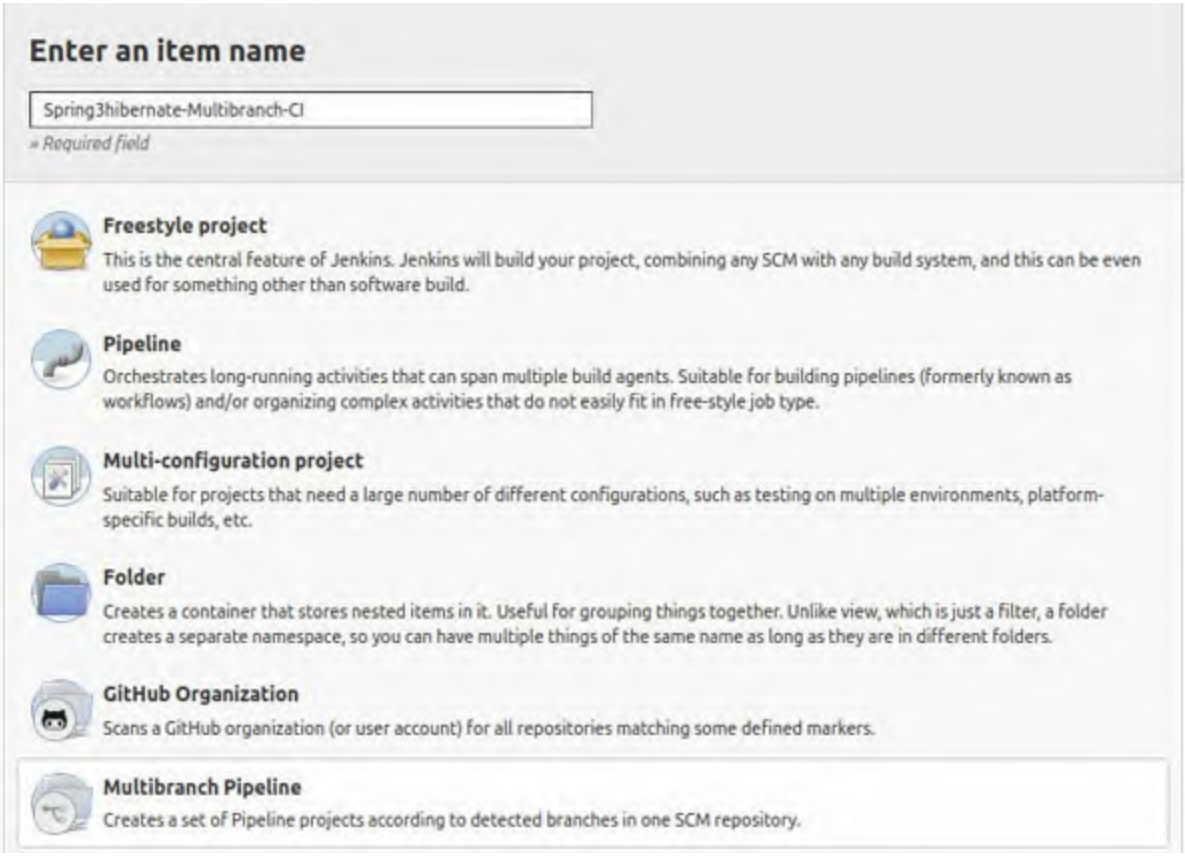

Figure 4.20: *Multibranch pipeline creation*

Provide the git repository information like this:

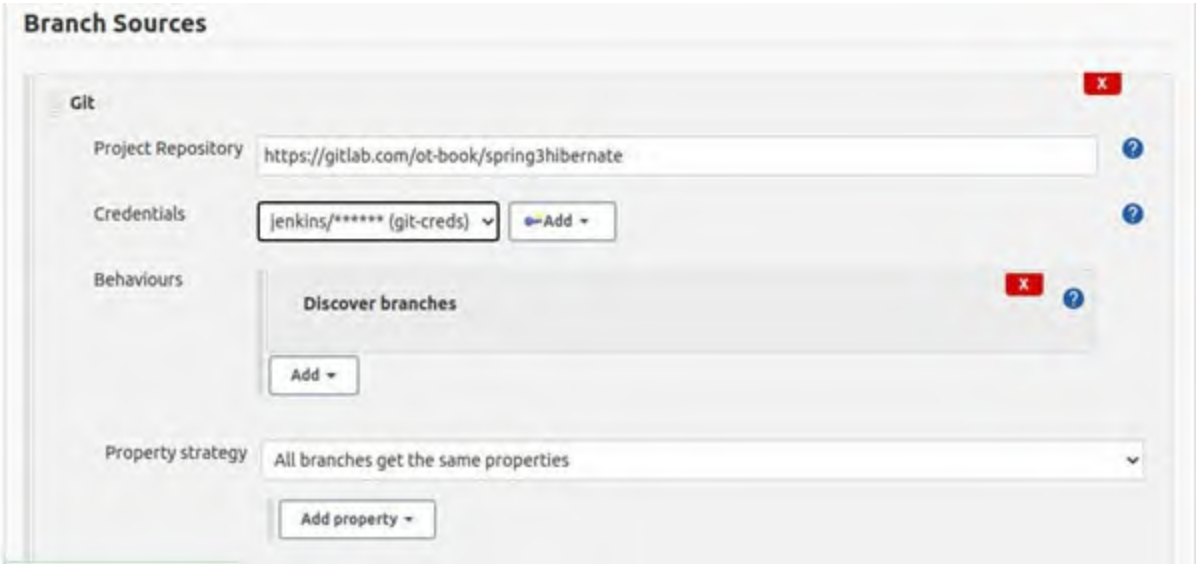

Figure 4.21: *SCM Configuration for Multibranch*

Save the pipeline and click on "**`Scan Multibranch Pipeline Now`**", as shown here:

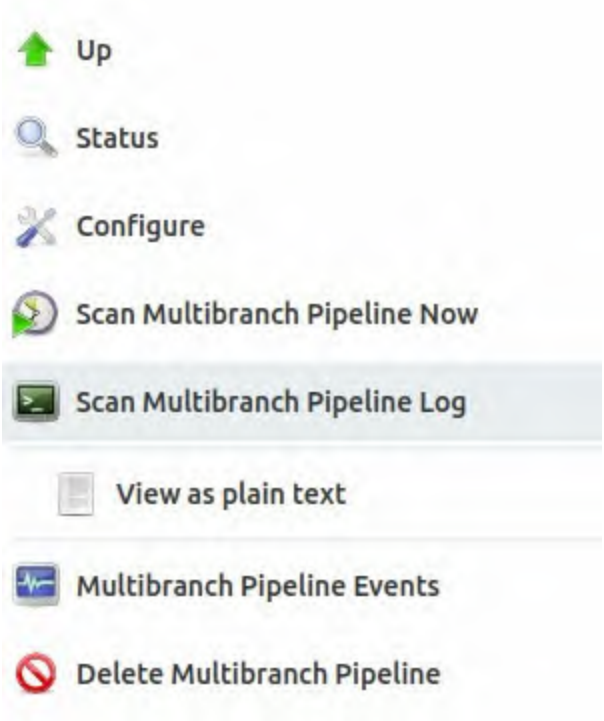

Figure 4.22: *Multibranch Pipeline options*

After the scan is complete, we will have a multi-branch pipeline like this:

Figure 4.23: *Multibranch Pipeline*

CI Pipeline update with Intermediate steps

After successfully creating the CI pipeline, I gave some internal demo to

Adeel, Sajal, and other colleagues. They liked the pipeline quite a lot. Although the general flow of the pipeline is complete, certain relevant intermediate steps are still missing. The steps that need to be added are listed here. They provide a segue between the CI and CD:

- Artifact Generation
- Artifact Upload to Nexus
- Deployment to Dev Environment
- DB Update

Now is the time to update our pipeline to accommodate these changes. But before executing these steps in Jenkins, we must do it manually in the dev environment. As mentioned earlier, the purpose of doing everything manually first is to perform a comprehensive study (POC) and have a detailed understanding of the topic. This will help in automating optimally.

It's quite straightforward. First, we need to generate the artifact on the local system before uploading it to the nexus repository, which we have already covered. We can generate the package(artifact) by executing the following code:

```
$ mvn clean package
```

Now, we have to upload this file to nexus. So, we have to create/update the **`~/.m2/settings.xml`** file with the following content:

```
<settings
xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.1.0
http://maven.apache.org/xsd/settings-1.1.0.xsd"
   xmlns="http://maven.apache.org/SETTINGS/1.1.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <servers>
   <server>
   <id>spring3hibernate</id>
   <username>${env.MAVEN_REPO_USER}</username>
   <password>${env.MAVEN_REPO_PASS}</password>
   </server>
   <server>
   <id>snapshots</id>
   <username>${env.MAVEN_REPO_USER}</username>
   <password>${env.MAVEN_REPO_PASS}</password>
   </server>
  </servers>
</settings>
```

Next, we can try uploading the artifact using maven cli, as follows:

```
$ export NEXUS_URL=http://<nexus_url>:8081
$ export MAVEN_REPO_USER=<nexus_user>
$ export MAVEN_REPO_PASS=<nexus_password>
$ mvn deploy
```

After the artifact is uploaded to nexus successfully, it's time to deploy the uploaded artifact on the dev environment. We already have an ansible playbook written for it in the repository and simply have to execute the ansible command line. The overview page of Nexus is shown in the following figure:

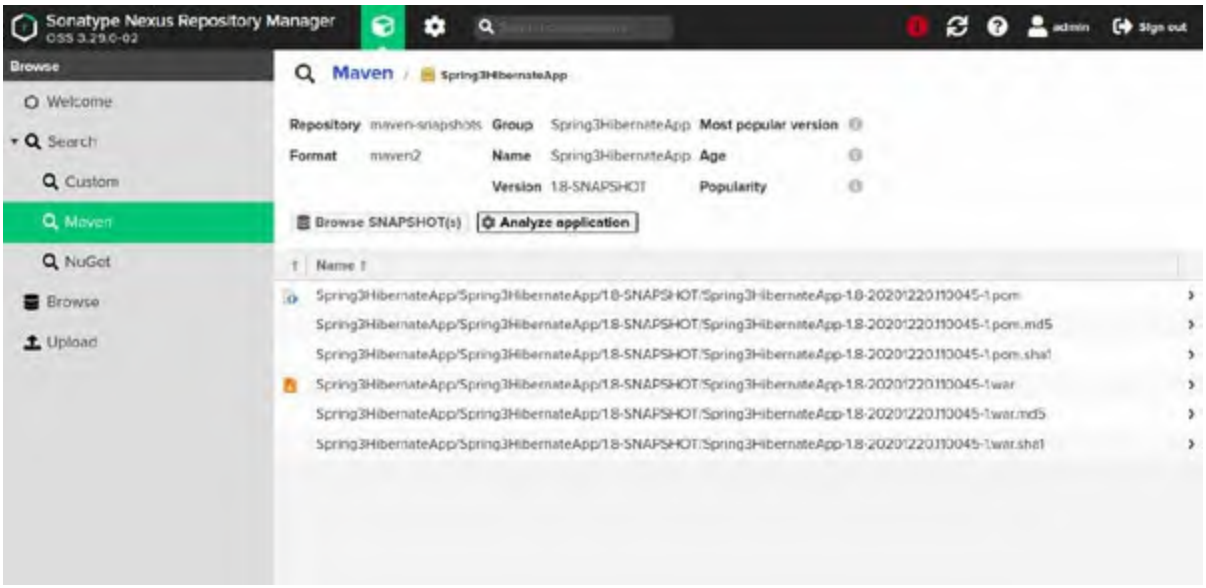

Figure 4.24: *Nexus Repository view*

Create a file named "**hosts**" and put this entry into the file with the following code:

```
[devservers]
devserver1 ansible_ssh_host=<dev_server_ip>
[devservers:vars]
ansible_ssh_user=<ssh_user>
ansible_ssh_pass=<ssh_password>
ansible_become_pass=<ssh_password>
```

Execute the playbook using the following command:

```
$ ansible-playbook -i hosts playbook.yaml -e
nexus_artifact_url="<package_url>"
```

Note: We are assuming that tomcat9 is already installed on the target dev environment server.

And in the last stage, we have to perform the DB update using flyway; since

flyway comes as an easy way to integrate the maven plugin, we simply have to execute it as follows:

```
$ mvn flyway:migrate
```

Figure 4.25 shows the flyway command output with the maven:

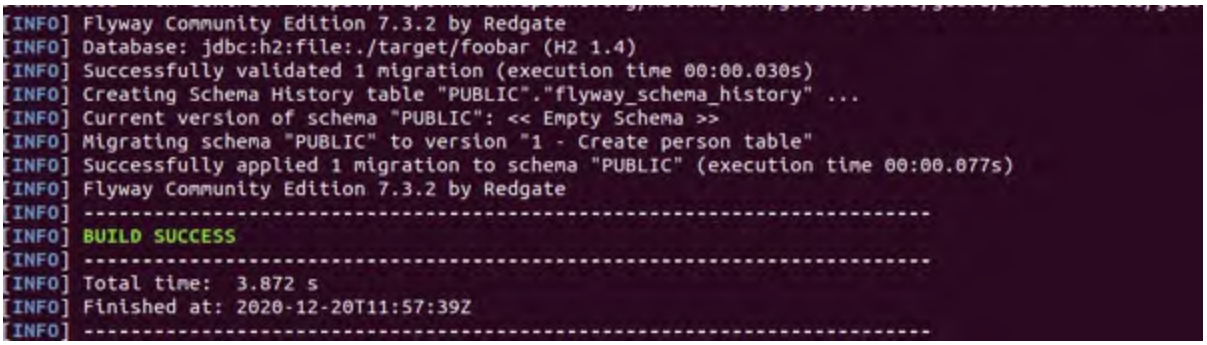

Figure 4.25: *Maven command output for flyway*

Now that we have executed all the steps on the dev system, it is time to integrate it with Jenkins.

Generating Artifacts

In the "*Code Stability*" stage of the pipeline, we are compiling the code that is actually generating an artifact that will be deployed to different environments. So, in this case, we do not have to repeat the same step.

Uploading Artifacts to Nexus

In the POC, we used maven to upload the war file on the nexus server, but with Jenkins, we will do it using the Jenkins nexus plugin. So, in any case, if there is a change in the nexus URL, we don't have to change the *pom.xml* file. Instead, we will make the changes in the Jenkins configuration.

First of all, we need to install the "**`nexus-artifact-uploader`**" plugin into the Jenkins system, as shown in the following figure:

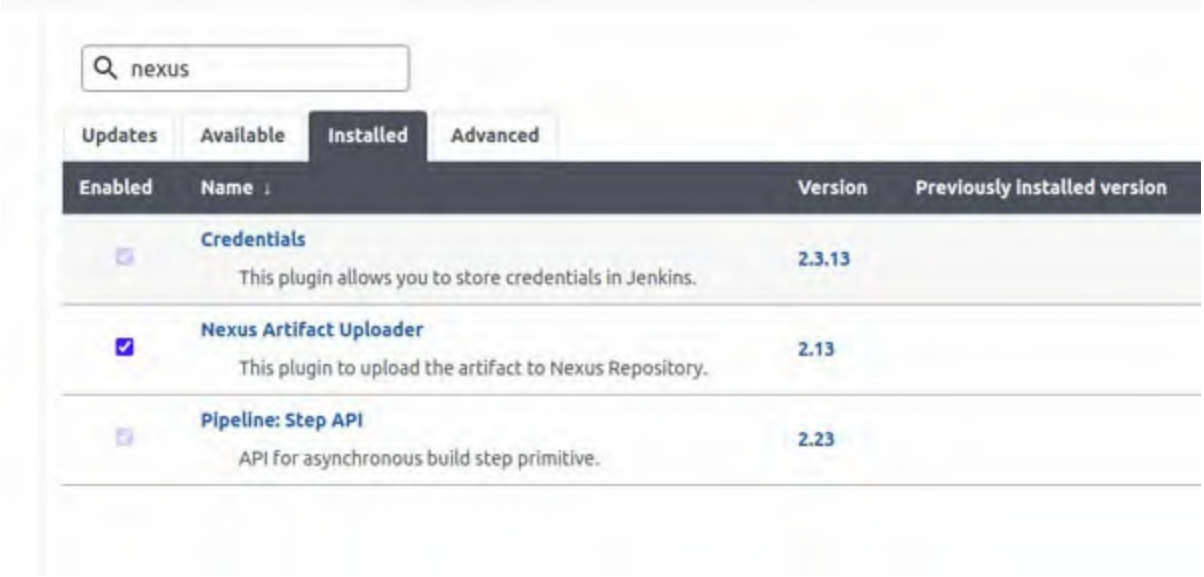

Figure 4.26: *Nexus Artifact plugin installation*

After that, we will add a credential called "**`nexus-creds`**" and provide a username and password for Nexus, as shown here:

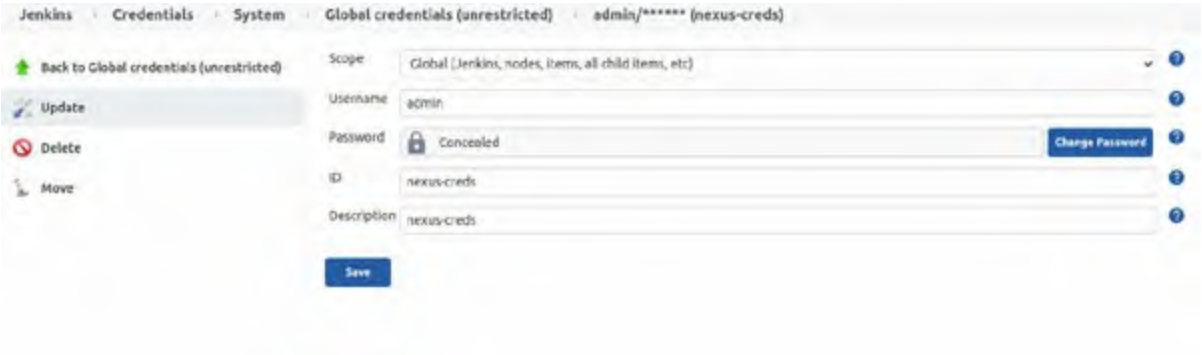

Figure 4.27: *Nexus credentials in Jenkins*

Once we are done with all these changes, we have to add a new stage in our pipeline called "**`Uploading artifact`**":

```
node("master") {
   stage("Checking out Code") {
     checkout scm
   }
   stage("Code Stability") {
     sh "mvn clean install"
   }
   stage("Code Quality") {
     sh "mvn checkstyle:checkstyle"
     recordIssues(tools: [checkStyle(pattern: '**/checkstyle-
     result.xml')])
```

```
    }
    stage("Unit Testing") {
      sh "mvn test"
      recordIssues(tools: [junitParser(pattern: 'target/surefire-
      reports/*.xml')])
    }
    stage("Security Testing") {
      sh "mvn org.owasp:dependency-check-maven:check"
      publishHTML([allowMissing: false, alwaysLinkToLastBuild:
      false, keepAll: false, reportDir: 'target', reportFiles:
      'dependency-check-report.html', reportName: 'Dependency
      Check Report', reportTitles: ''])
    }
    stage("Sonarqube Analysis") {
      sh "mvn sonar:sonar -Dsonar.host.url=${SONAR_URL} -
      Dsonar.login=${SONAR_USER} -
      Dsonar.password=${SONAR_PASSWORD} -Dsonar.java.binaries=."
    }
    stage("Uploading artifact") {
      nexusArtifactUploader artifacts: [[artifactId:
      'spring3hibernate', classifier: '', file:
      'target/Spring3HibernateApp.war', type: 'war']],
      credentialsId: 'nexus-creds', groupId: 'org', nexusUrl:
      '<nexus_url>:8081/', nexusVersion: 'nexus3', protocol:
      'http', repository: 'spring3hibernate', version: 'v0.1'
    }
}
```

Note: Change the <nexus_url> with the IP or domain name of your nexus server.

Then, simply add a snippet into Jenkinsfile and execute the job. As a result, we will have a stage like this:

Figure 4.28: *Stage view with uploading artifact*

Deployment to Dev Environment

As the ansible-playbook is ready for the deployment, the only change we have to make at the Jenkins side is to add a stage to deploy the code on the dev environment. The code for this is as follows:

```
node("master") {
   stage("Checking out Code") {
    checkout scm
   }
   stage("Code Stability") {
    sh "mvn clean install"
   }
   stage("Code Quality") {
    sh "mvn checkstyle:checkstyle"
    recordIssues(tools: [checkStyle(pattern: '**/checkstyle-
    result.xml')])
   }
   stage("Unit Testing") {
    sh "mvn test"
    recordIssues(tools: [junitParser(pattern: 'target/surefire-
    reports/*.xml')])
   }
   stage("Security Testing") {
    sh "mvn org.owasp:dependency-check-maven:check"
    publishHTML([allowMissing: false, alwaysLinkToLastBuild:
    false, keepAll: false, reportDir: 'target', reportFiles:
    'dependency-check-report.html', reportName: 'Dependency
    Check Report', reportTitles: ''])
   }
   stage("Sonarqube Analysis") {
    sh "mvn sonar:sonar -Dsonar.host.url=${SONAR_URL} -
    Dsonar.login=${SONAR_USER} -
    Dsonar.password=${SONAR_PASSWORD} -Dsonar.java.binaries=."
   }
   stage("Uploading artifact") {
    nexusArtifactUploader artifacts: [[artifactId:
    'spring3hibernate', classifier: '', file:
    'target/Spring3HibernateApp.war', type: 'war']],
    credentialsId: 'nexus-creds', groupId: 'org', nexusUrl:
    '<nexus_url>:8081/', nexusVersion: 'nexus3', protocol:
    'http', repository: 'spring3hibernate', version: 'v0.1'
   }
   stage("Deploying to Dev Environment") {
    sh "ansible-playbook -i hosts playbook.yaml -e
    nexus_artifact_url=<artifact_url>"
   }
```

```
}
```

Note: Change the <artifact_url> with the actual artifact URL.

The stage view with deployment on the dev environment is shown in the following figure:

Figure 4.29: *Stage view with Dev environment deployment*

DB Update

DB update is an important step to manage schema changes inside the database so that the new version of the application does not crash or fail because of the unfamiliar database schema.

Finally, the last intermediate step is "*DB Update*," which can be done using maven, as follows:

```
node("master") {
   stage("Checking out Code") {
     checkout scm
   }
   stage("Code Stability") {
     sh "mvn clean install"
   }
   stage("Code Quality") {
     sh "mvn checkstyle:checkstyle"
     recordIssues(tools: [checkStyle(pattern: '**/checkstyle-
     result.xml')])
   }
   stage("Unit Testing") {
     sh "mvn test"
     recordIssues(tools: [junitParser(pattern: 'target/surefire-
     reports/*.xml')])
   }
   stage("Security Testing") {
```

```
        sh "mvn org.owasp:dependency-check-maven:check"
        publishHTML([allowMissing: false, alwaysLinkToLastBuild:
        false, keepAll: false, reportDir: 'target', reportFiles:
        'dependency-check-report.html', reportName: 'Dependency
        Check Report', reportTitles: ''])
    }
    stage("Sonarqube Analysis") {
        sh "mvn sonar:sonar -Dsonar.host.url=${SONAR_URL} -
        Dsonar.login=${SONAR_USER} -
        Dsonar.password=${SONAR_PASSWORD} -Dsonar.java.binaries=."
    }
    stage("Uploading artifact") {
        nexusArtifactUploader artifacts: [[artifactId:
        'spring3hibernate', classifier: '', file:
        'target/Spring3HibernateApp.war', type: 'war']],
        credentialsId: 'nexus-creds', groupId: 'org', nexusUrl:
        '<nexus_url_or_ip>:8081/', nexusVersion: 'nexus3', protocol:
        'http', repository: 'spring3hibernate', version: 'v0.1'
    }
    stage("Deploying to Dev Environment") {
        sh "ansible-playbook -i hosts playbook.yaml -e
        nexus_artifact_url=http://<nexus_url_or_ip>:8081/repository/s
        v0.1.war"
    }
    stage("DB Update") {
        sh "mvn flyway:migrate"
    }
}
```

Note: Replace the nexus_url_or_ip with the valid nexus URL.

After adding all the intermediate stages successfully, we will have a resultant stage view like this:

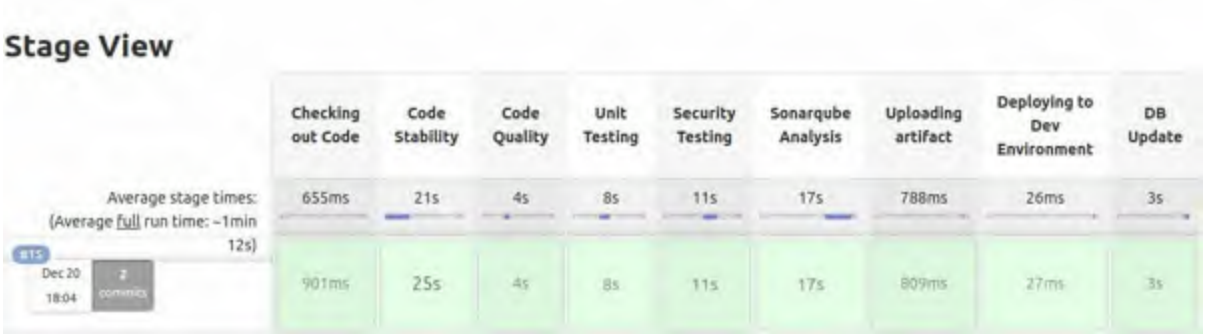

Figure 4.30: *Stage view with DB update*

CI Pipeline with Notification Integration

Now, this is more of a convenience than an actual step in the CI/CD pipeline. We might need notifications at various stages, that is, at the start of the build, at successful execution, at failed execution, when the job is aborted, and so on. It could be through email or various platforms that can be integrated with Jenkins, like slack, hipchat, and so on. Here, we will go with both email and slack as they are the ones most commonly used. Assuming that slack and email configurations are already done on the Jenkins server, the following code is the example of notification stage along with other CI steps:

Running the pipeline will produce the following stage view:

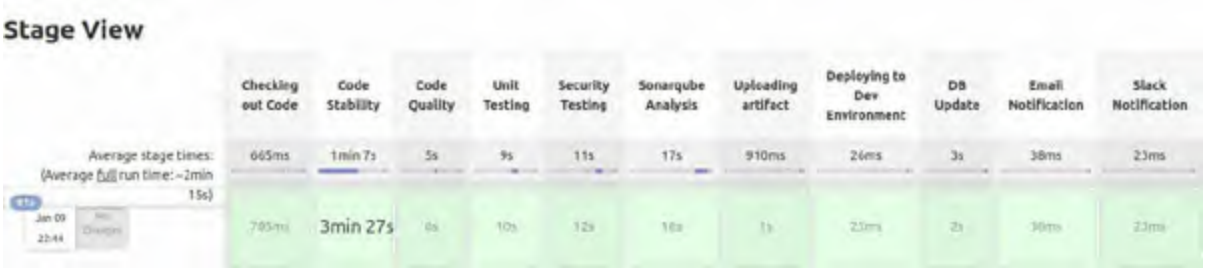

Figure 4.31: *Stage view with Slack Notification*

For more information about configuring email and slack in the Jenkins system, go through the following reference documentation:

https://plugins.jenkins.io/slack/

https://plugins.jenkins.io/email-ext/

Conclusion

Everything worked as I had hoped it would. I know it’s a dream in most cases to be an IT professional. Then again, it wasn’t my first time. The difficult part is always planning and research. Post that, what’s left is to follow the links, one step after the other; play around and see what possibilities we have; and explore them to find out what works together and what doesn’t. This is how we can come close to designing automation with “Good Taste” in it. If you are wondering what I mean by “Good Taste”, google “Linus Torvalds Good Taste” and find out. After this successful implementation, I did the same with Adeel’s team on different company projects. We hit some bumps on the way, but nothing tenacious. It worked well. I was looking forward to getting a chance to solve more issues as they always open up new opportunities.

In the next chapter, we will discuss an interesting pattern of problems emerging post-CI implementation.

Questions

1. What is a multi-branch pipeline?
2. How was the HTML report of security testing useful?
3. How can we configure artifact upload?
4. What are the stages in a pipeline?

CHAPTER 5

Introduction to Docker

The implementation of CI was successful, and we could fix almost every issue people faced in delivering the software. However, after some time, the operations and development teams raised new issues. The biggest challenge was a conflict between the teams, "*It is working on my system.*" Sometimes, the application deployment in the production environment was a failure. Still, when the operations team checked with the development team, they saw the application working fine on their system and considered it an infrastructure issue. The operations team returned with the point that we had not changed anything on the infrastructure, so this must be a code issue. After endless hours of discussions and debugging, the teams identified many discrepancies between the development and production environments. In this chapter, let us see how these issues got resolved.

Structure

In this chapter, we will discuss the following topics:

- Need for containerization
- What and why containers?
- Container engines
- Docker installation

Objectives

After going through this chapter, you should understand the problem of platform dependency. We will also talk about the need for containers and related concepts. We will understand how Docker works as a modern container engine and what its architecture and components are.

Need for containerization

Sajal: “Guys, it seems like we have faced a lot of production downtimes in recent days. Does anyone know what the cause could be and how we can prevent such downtimes in the future?”

Adeel: “We have done thorough testing while developing and verified whether each service/component is working correctly.”

Sonia: “All the tests have been passed in the QA testing, and we have historical data available regarding the report.”

Me: “We have not changed anything on the infrastructure side, so I do not think it is an infrastructure issue.”

Sandeep: “Still, we faced the issues while releasing the code, right? Also, despite the code being released, we were getting bugs and issues reported from everywhere. There has to be a valid explanation for this.”

Me: “I have one theory about this. I think it could be an infrastructure/platform discrepancy issue because we all use different mechanisms to build, test and deploy code. Right now, the development team is using Windows OS for development. Similarly, in the QA environment, there is a mix of people using the Windows OS and Ubuntu (Linux) OS. While releasing the code, we are releasing it on a CentOS-based OS.”

Sajal: “This is a very valid point. We had not noticed that we are using different OS and platforms in each environment.”

Sandeep: “So what do you suggest? Shall we keep the same OS for all environments?”

Me: “I do not think that will be a good idea because then, people have to learn other OS for their work, which will impact efficiency and comfort. I would suggest that we take a new approach, called ‘containerization’.”

Sajal: “But how is this going to help us?”

Sandeep: “Before answering Sajal’s question, I would like to know what exactly containerization means?”

Me: “Containerization is a way to encapsulate or isolate our application and its dependencies so that it can be run on any infrastructure. For example, you must have seen that in the shipping industry, they encapsulate the same kind of material in a container and then use the same container to ship it everywhere. In that case, it doesn’t matter if the container is on a ship/truck or a cargo plane.

Now coming to Sajal's question, we will use the container philosophy to isolate our CI stages so that they can become OS-independent. For each step, we will use the containers so that there will be no discrepancies between the build, test, and deployment stages."

Sajal: "So, you mean all the CI steps will be performed inside a container so that we will use the same library and dependencies everywhere?"

Me: "Exactly! We will do the same thing using the container technology. This will not only help us in removing the discrepancy but will also help in the prevention of OS failure of the Jenkins server."

Adeel: "Whoa! This seems interesting. How will the container help us in such scenarios?"

Me: "Everything will happen on individual containers and software. So, if anything wrong happens, it will occur on the container only. The OS and the Jenkins server will not have any impact."

Sandeep: "This sounds promising; I think we should use this approach. Have you finalized a container solution for the same?"

Me: "Let me study and compare the available solutions of containers before choosing from them."

Sajal: "Sounds like a plan!"

What and why containers?

Before jumping into the discussion of what containers are, we need to understand the need for container technology in the modern world. Before that, we need to understand the concept of virtualization.

Virtualization

Virtualization refers to running multiple virtual systems or resources on a physical machine. It allows us to create a virtual machine in a layer abstracted from the actual hardware. There are a few essential things in a virtual machine:

- **Hypervisor:** A hypervisor is a program or OS used for creating, running, and managing virtual machines. Our modern cloud uses the concept of Hypervisor.

- **Virtual Machines:** A virtual machine is the emulated equivalent computer system running on top of another computer. Virtual machines may have minimal access to the host system's resources.

Here is the virtualization architecture:

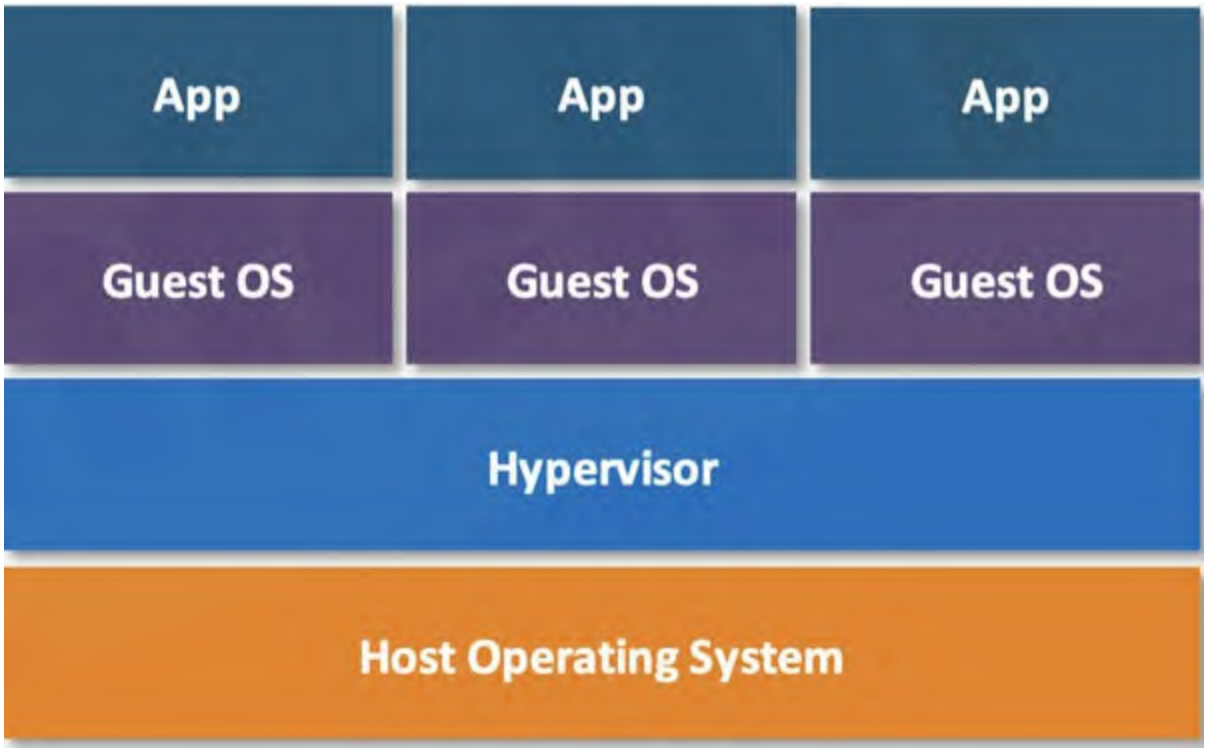

Figure 5.1: *Virtualization Architecture*

Virtualization takes away a lot of pains in infrastructure management. However, there are a few limitations of infrastructure management using virtualization:

- The server cost is high because we need to set up each application on a different server to ensure the isolation of the process.
- If we are setting up a single application on the physical machine, the resources will not be utilized properly.
- Disaster recovery is difficult when hardware fails because all the data will be lost.
- The process of getting any software up and running is time-consuming.

What is a Container?

Containerization is an OS-level virtualization method used to deploy and run distributed applications without launching an entire **Virtual Machine** (**VM**) for each application.

It is a kind of OS virtualization where we run our applications in a separate user space called containers. Here are a few properties of containers:

- They have everything that is necessary to run an application, like OS library, binaries, dependencies, and configuration files, except the kernel.
- We can also say that a container is a lightweight virtual machine with limited access to the host OS.
- We can run containerized applications anywhere, without depending on the infrastructure, i.e., it can be run on a private data center machine or on a VM in the cloud.

An architecture diagram of a container looks as follows:

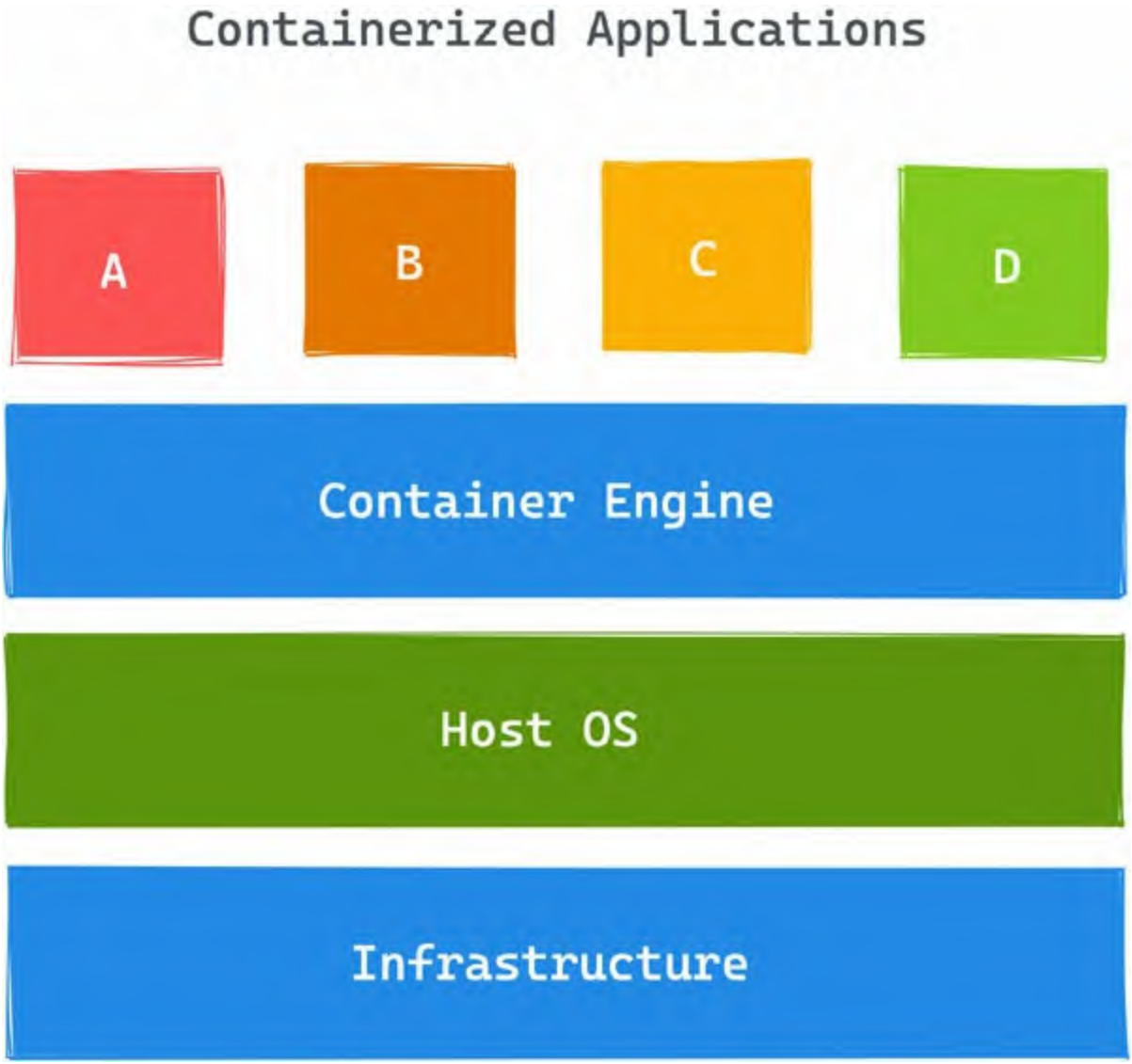

Figure 5.2: *Container Architecture*

Why Container?

Containerization solves a lot of problems that people face with virtualization. The key highlights are mentioned here:

- **OS Dependency:** While working the containers, we are not at all dependent on the OS and the architecture of the base OS.
- **Resource Optimization:** Virtual machines are heavier than containers because of their kernel and all bootup files. Since containers only have binaries, libs, and configurations, they are more beneficial in resource management.

- **Deployment:** Deployment time is high on virtual machines because of their process bootup and configuration time; on the other hand, container deployment is quick.
- **Environment Consistency:** Deploying applications in different environments is consistent because containers are not affected by the underlying infrastructure and OS.

Container Engines

When we talk about containers, people generally think “Docker” and assume that Docker is the container, but it’s not like that. Containerization is a technology, and Docker is a provider of containerization, just like Linux is an OS architecture and Debian and RedHat are providers.

The following image shows multiple container engine (runtime) providers available:

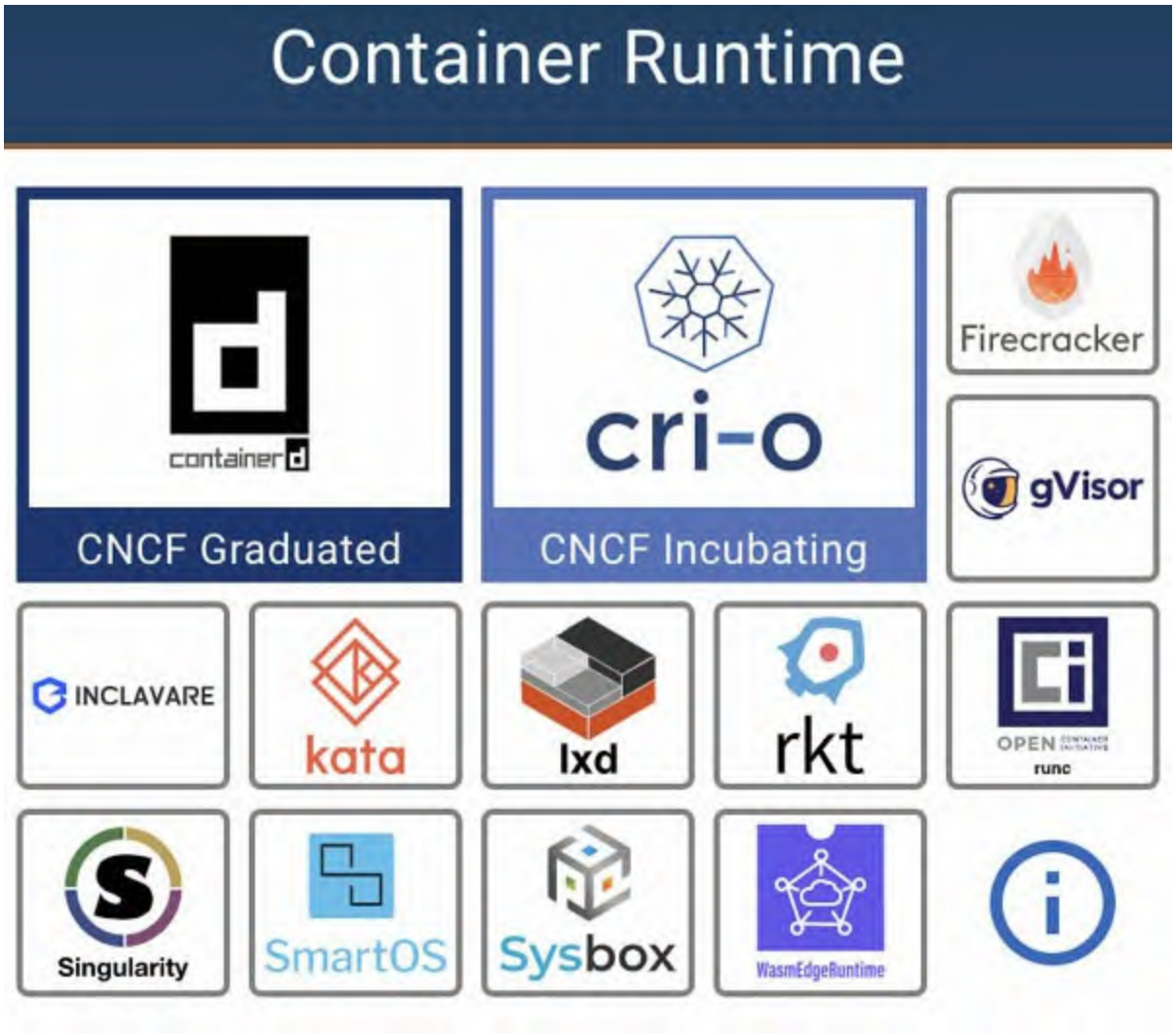

Figure 5.3: *Container Engines*

The preceding figure shows the following:

- CRI-O
- ContainerD
- Kata
- Firecracker

Docker is also a container provider based on container D runtime, which is CNCF graduated project. It has excellent community support, with more than 60k+ GitHub stars and 18k+ developers. Many famous companies have adopted Docker as their container engine and are running their production workload on it.

Docker Basics

Docker containers wrap up a piece of software in a complete filesystem that contains everything it needs to run:

- Code
- Runtime
- System tools
- System libraries

The Docker is an open platform to build, ship, and run distributed applications. It will always run the same, regardless of the environment it is running in. Also, it is a subset project of moby (**https://github.com/moby/moby**). The architecture of Docker is described as follow:

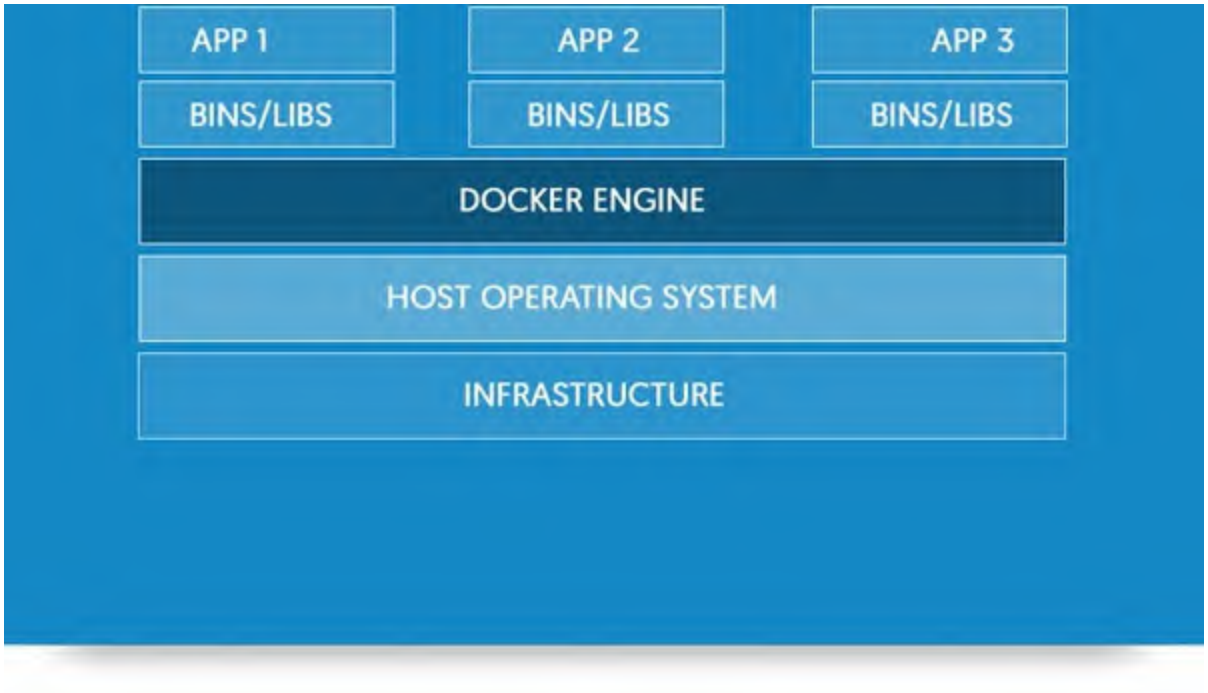

Figure 5.4: *Docker Architecture Diagram*

Docker architecture

The Docker architecture is client-server-based architecture. The Docker client communicates with the Docker daemon, which builds, runs, and distributes Docker containers. You can run a Docker client on the same system as the

Docker daemon, or you can connect a Docker client to a Docker daemon running remotely. Refer to the following figure:

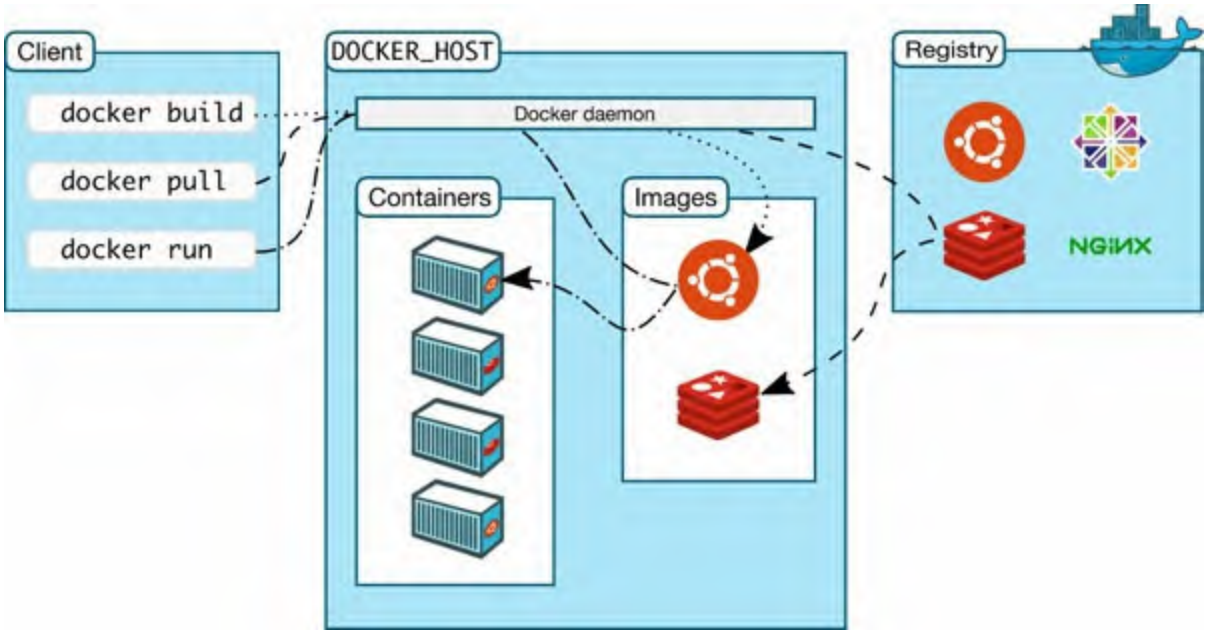

Figure 5.5: *Docker Engine Architecture*

Docker Images

Docker image is a definition from which containers can be created. Images are inherited from base images and can be many levels deep. They are immutable in nature, which means they cannot be changed; if we want to change a docker image, we have to create a new one.

Docker images work in layer mode; every step of execution will create a new layer inside an image. Once the image is created, we can store it on the remote repository for further processing. When pushing/pulling an image to/from a repository (more to come), only the changed layers are pushed/pulled to save bandwidth.

There are two ways of creating images inside Docker:

- Docker commits
- Dockerfile

You can see the differences illustrated in the following table:

Docker Commit	Dockerfile

Docker commit takes a snapshot of a running container.	Dockerfile is a set of instructions through which a docker image can be created.
It does not compare and detect the changes, i.e., a new image will be created every time.	Every time we build an image using a docker file, the changes will be compared.
Image size will not be consistent when using this method. The size will reduce if we delete something inside the container and vice versa.	Image size will be consistent because there are fewer chances of manual changes.
It cannot be stored in the form of code.	It will be stored in the form of a file that can be versioned on VCS.

Table 5.1: *Comparison between commit and Dockerfile*

Command reference for “Docker commit”:

```
$ docker commit <container-id>
```

Code 5.1

Command reference for building a Dockerfile:

```
$ docker build -t <image_name>:<image_tag> -f Dockerfile.
```

Code 5.2

Dockerfile

A Dockerfile contains all the commands a user can run on the command line to assemble an image. Docker build allows users to automate the execution of several commands at once using multiple command-line instructions. In figure 5.6, a sample Dockerfile is shown:

Figure 5.6: *Sample Dockerfile*

`FROM` is to call a base image from the registry; it should be a valid image name.
`COPY` is a step that is used to copy content from the local filesystem to an image.
`WORKDIR` is used to set the working directory in which other Dockerfile steps will be performed.
`RUN` is to execute any command inside the docker image.
`ENTRYPOINT` allows us to run executable files or commands, like Java and node. It is the main command that allows a Docker container to run a process.
`CMD` is an instruction that can be executed once after the Dockerfile is built, generally, people use CMD to pass arguments to ENTRYPOINT.

There are some other important instructions available in the Dockerfile, which can be used in different scenarios:

- **`USER`** is an important instruction in a Dockerfile that can be used to change the user of the docker image; whenever a container is started, it will be started with the defined user. If this parameter is not defined, the

default root user will be used.

- **`VOLUME`** is a filesystem mounted on Docker containers to preserve data generated by the running container.

For more information, refer to the official documentation of Dockerfile at the following link:

https://docs.docker.com/engine/reference/builder/

Multistage Dockerfile

A Dockerfile multi-stage build is a separate build as compared to the runtime environment to reduce the image size and dependencies. It allows a slight variation of changes in Docker images.

By using this method, images can be built platform-specific with minimal software and dependencies. It also helps in linearizing the dependencies for the application.

A simple example could be this: for compiling a Java-based application, we need Maven, but for running the application, we don't need JDK and maven. Refer to the following figure:

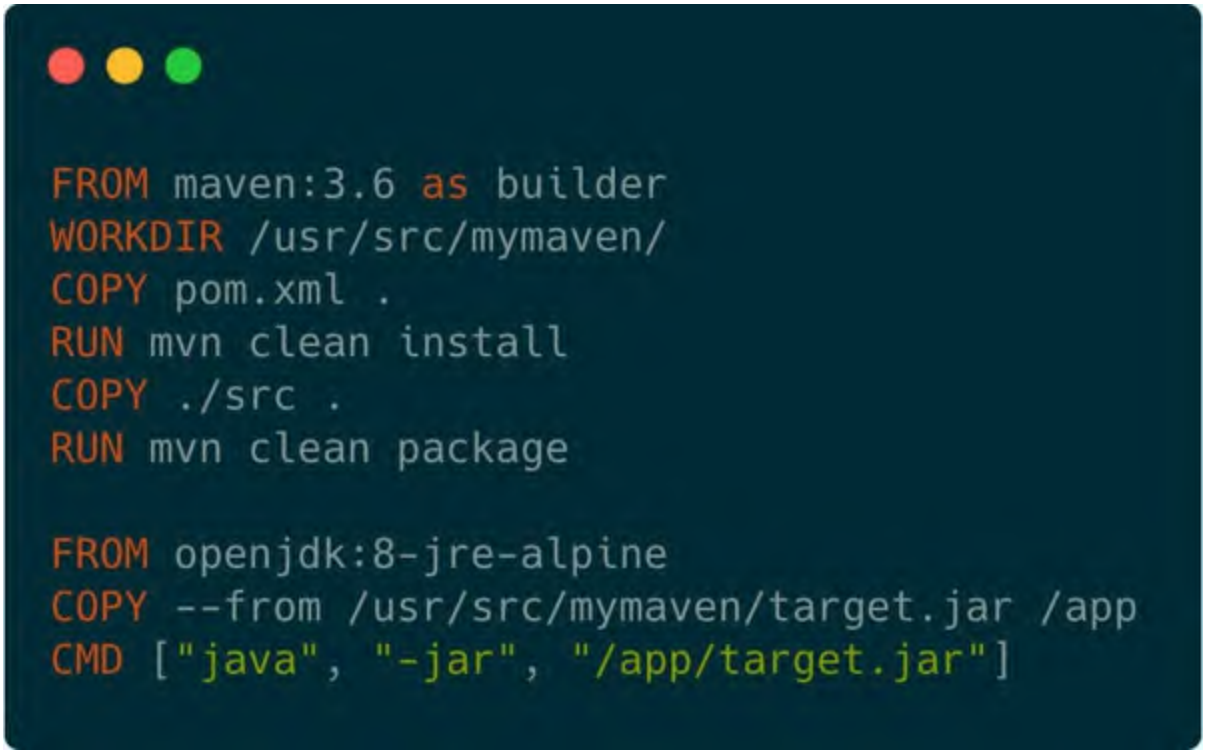

Figure 5.7: *Multistage Dockerfile*

In the preceding example, we can compile the application in the first stage for generating the artifacts; then, we can copy the generated artifacts to the runtime image, which will have minimal packages and dependencies.

So, in a multistage Dockerfile, each stage starts “FROM,” where it pulls a new base image and copies the binaries from the other image’s stage to this new base image.

Also, while copying the package and binaries, we ensure that we only include the essential packages that are needed to run the application.

Docker Registry

Docker registries store and distribute named Docker images. There may be multiple versions of the same image, identified by their tags. By default, all images are pushed and pulled from the official docker registry called “DockerHub”.

To push the images to a different location or repository, we may have to append the URL before the name and tag of the Docker image. This tells us from where we have to pull and push images. Consider the following example:

quay.io/opstree/spring3hibernate:v1

If a registry is secured, we may have to log in to it using the **`docker login`** command.

The architecture of Docker registry is shown as follow:

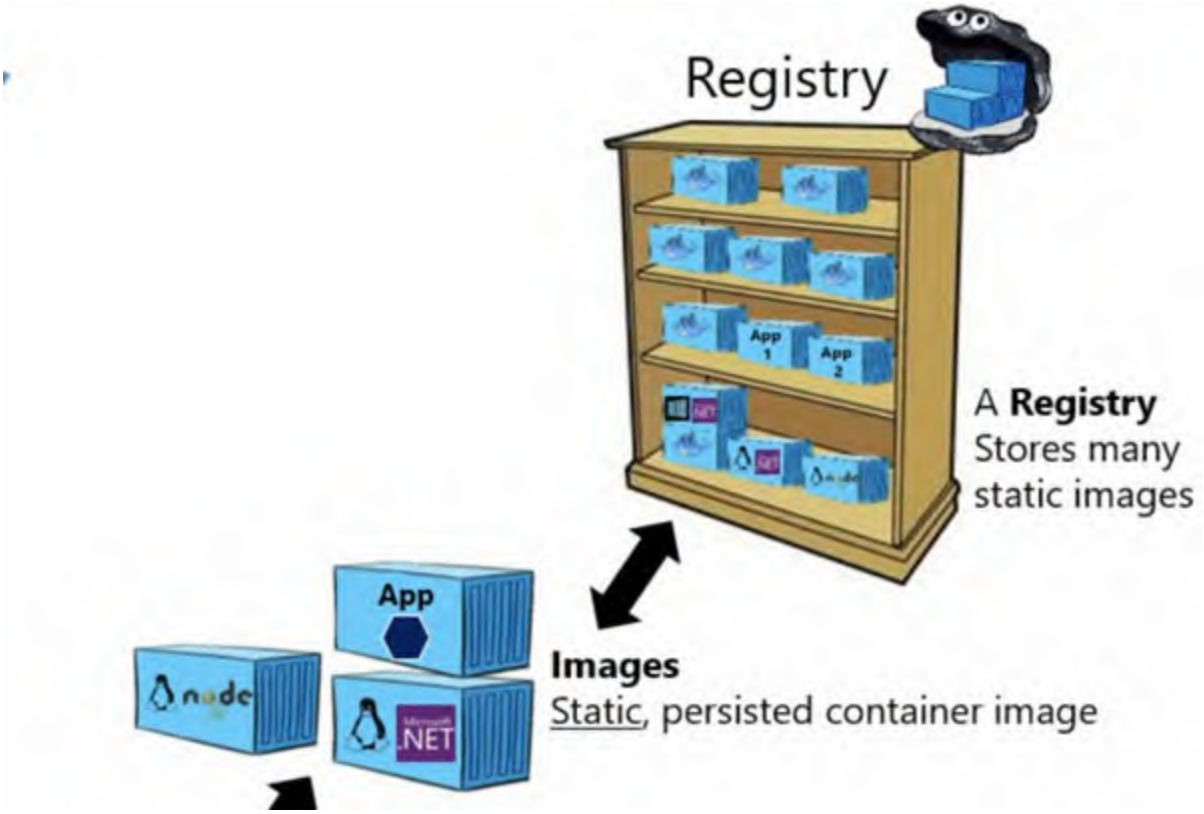

Figure 5.8: *Multistage Dockerfile*

Docker CLI

Using Docker's command-line interface, we can build, interact, inspect, and run containers and images. Many of the commands are similar to Linux commands. All the commands will be started using "*docker*".

For every command, we have a help page available inside the terminal; consider the following example:

```
$ docker run --help
```

Code 5.3

In the following table, different Docker command arguments are explained:

Command	Description
`ps`	To list running containers (include –a to see stopped containers)
`rm`	To remove containers from the system
`inspect`	To return information about the docker objects, like images and containers

`start/ stop/ restart`	To start, stop or restart a container, respectively
`cp`	To copy files/folders between a container and the local filesystem and vice versa
`build`	To build an image from a Dockerfile
`images`	To list images
`rmi`	To remove one or more images from the system
`push/pull`	To pull and push Docker images from the remote repository
`tag`	To tag the images

Table 5.2: *Docker commands and their descriptions*

Docker Installation (Debian System)

To install Docker Engine for the first time on a new host machine, you must first set up the Docker repository. Docker can then be installed and updated from the repository.

Install packages to allow apt to use HTTPS repositories in the apt package index:

```
$ sudo apt-get update
$ sudo apt-get install \
  ca-certificates \
  curl \
  gnupg \
  lsb-release
```

Code 5.4

Add the docker GPG key in the system:

```
$ sudo mkdir -p /etc/apt/keyrings
$ curl -fsSL https://download.docker.com/linux/debian/gpg | sudo
gpg --dearmor -o /etc/apt/keyrings/docker.gpg
```

Code 5.5

We will use this command to set up the Debian repository:

```
$ echo \
  “deb [arch=$(dpkg --print-architecture) signed-
  by=/etc/apt/keyrings/docker.gpg]
  https://download.docker.com/linux/debian \
  $(lsb_release -cs) stable” | sudo tee
  /etc/apt/sources.list.d/docker.list > /dev/null
```

Code 5.6

Next, we need to update the apt package repository and install the Docker engine:

```
$ sudo apt-get update
$ sudo apt-get install docker-ce docker-ce-cli containerd.io
docker-compose-plugin
```

Code 5.7

The final step would be verifying the docker installation using the *docker* command:

```
$ sudo docker run hello-world
```

Code 5.8

Conclusion

I am hopeful that we'll be able to solve the conflict between the teams. Once the learnings from this study are compiled into the proposal, I'll present it to the others and gather their feedback. After everyone is onboard, we can quickly start working together to nip this in the bud. I might be getting ahead of myself, but it is hard not to imagine a favorable outcome. Stories of the past tell us how crucial and beneficial it has been for teams to move to containers. The development team gets freedom, and the operations team can focus on a planned infrastructure enhancement instead of having to play catch-up and extinguish fires.

In the next chapter, we will focus on CI with both Jenkins and Docker. We will use Docker to containerize our microservice with the best standardization and security practices.

CHAPTER 6

CI with Jenkins and Docker

After the explanation by Abhishek, all the team members evaluated the container technologies by reading books, journals, and white papers about them. Since the problem was evident in the heads of all team members, they mutually agreed to use Docker for continuous integration. Also, the development team decided that they would containerize their applications in the next phase. But again, before implementing this pipeline and process, they need a POC with Docker's CI steps. That's why the team started to build the Docker process to take their Continuous Integration process to the next level.

Structure

In this chapter, we will discuss the following topics:

- Containerization of application
- CI pipeline with pre-deployment
    - Code stability
    - Code quality
    - Unit testing
    - Code coverage
    - Security testing

Objectives

After going through this chapter, you should be able to understand the concept of Jenkins with the Docker engine. You should also be able to understand all CI steps as a part of a different Docker container. Pipeline creation of Jenkins with Docker and docker working as a slave with Jenkins will also be discussed in this chapter.

Containerization of application

As an initial stage of containerization, we need to write a **`Dockerfile`** for our application. Once the Dockerfile is ready, we can build the image from it, and the image can be transported to any environment or system.

The Dockerfile for our application will look like this:

```
FROM maven:3.3-jdk-8 as builder
COPY pom.xml /usr/src/spring3hibernate/
COPY ./src /usr/src/spring3hibernate/
WORKDIR /usr/src/spring3hibernate/
RUN mvn clean install && \
   mvn package
FROM tomcat:7-jre7-alpine
MAINTAINER “opstree <opstree@gmail.com>”
RUN rm -rf /usr/local/tomcat/webapps/*
COPY --from=builder
/usr/src/spring3hibernate/target/Spring3HibernateApp.war
/usr/local/tomcat/webapps/ROOT.war
WORKDIR /usr/local/tomcat/webapps/
EXPOSE 8080
```

Code 6.1

Now, we can quickly build the application image from the Dockerfile using the docker plugin inside Jenkins, as shown in the following figure:

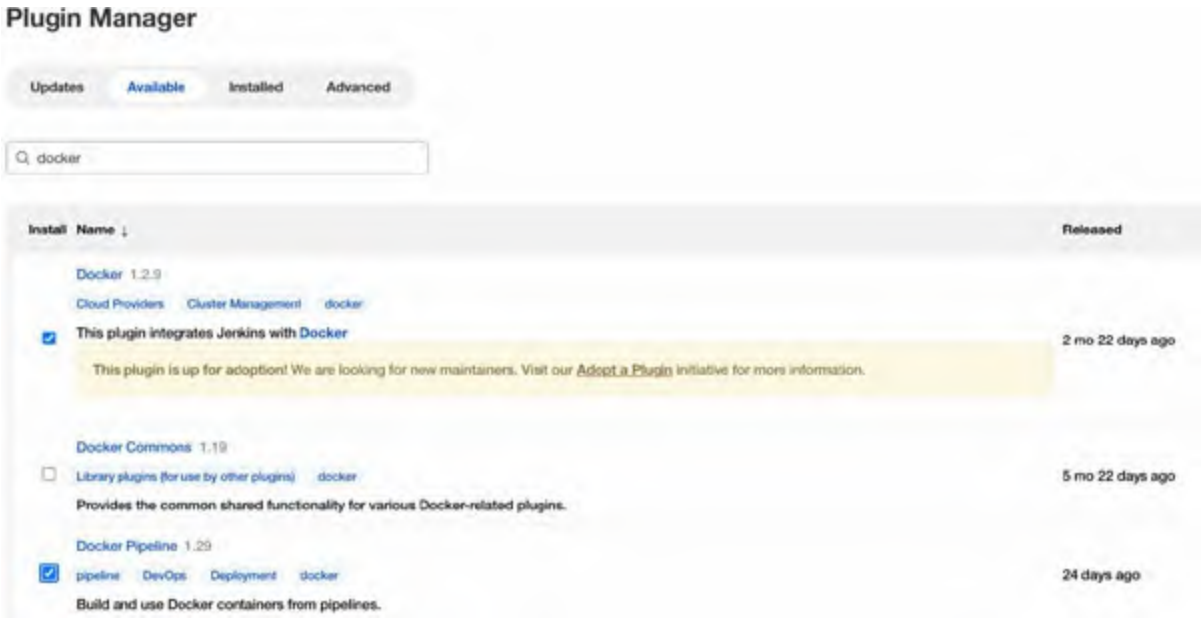

Figure 6.1: *Docker Plugin*

CI Pipeline with Pre-Deployment Integration Checks

Before executing all the pre-deployment steps, checks using Jenkins and Dockerfile local validation are essential. So, let's compile this project over Docker to see if containerization is working correctly:

```
$ docker build -f Dockerfile -t
quay.io/opstree/spring3hibernate:v1 .
```

Code 6.2

Figure 6.2 shows the complete output of the docker build command:

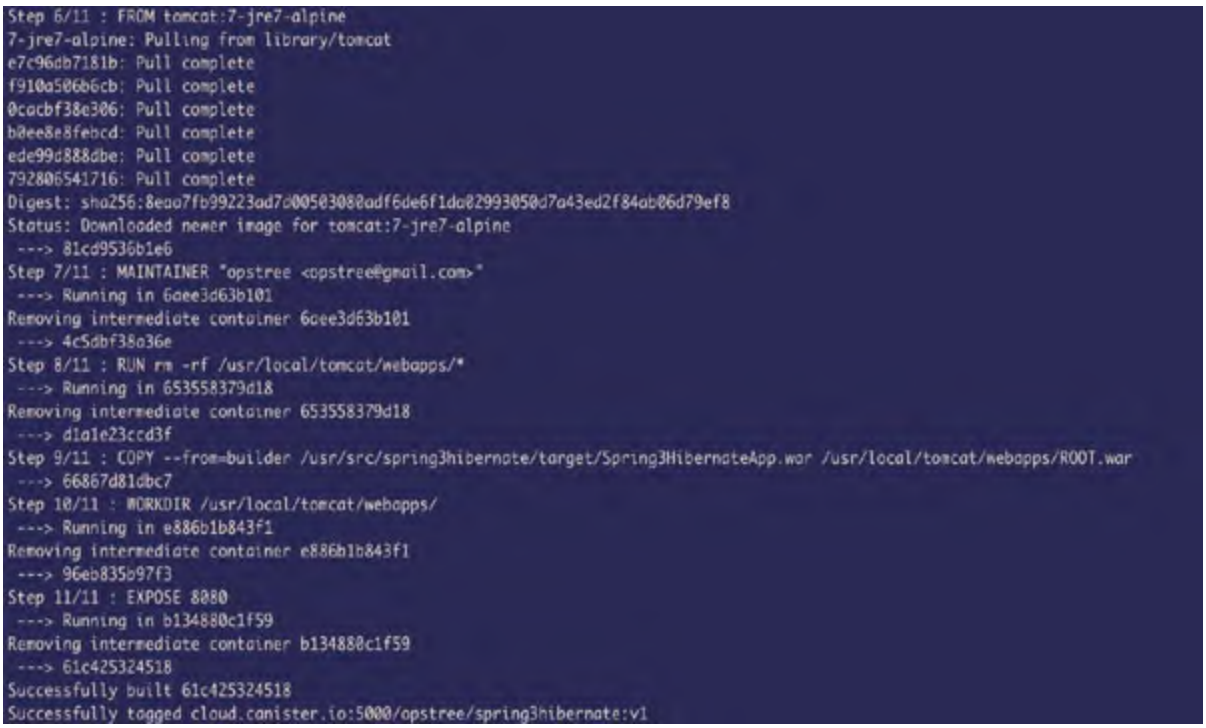

Figure 6.2: *Docker Image*

Code Stability

After successfully validating the Dockerfile for the application, we can start integrating the CI pipeline with Jenkins and Docker. The steps for code stability will be similar to prior implementations. Still, since our application is running as a container, we need to check whether the Docker image is getting built successfully at this stage.

The pipeline snippet for Code Stability will look like this:

```
pipeline {
```

```
  agent any
  stages {
     stage("Code Checkout") {
     steps {
      git credentialsId: 'git-creds', url:
      'https://gitlab.com/ot-book/spring3hibernate.git'
     }
     }
     stage("Code Stability | Build Image") {
     steps {
      script {
       docker.build("cloud.canister.io:5000/opstree/spring3hibern
      }
     }
     }
  }
}
```

Code 6.3

The stage view will look like this:

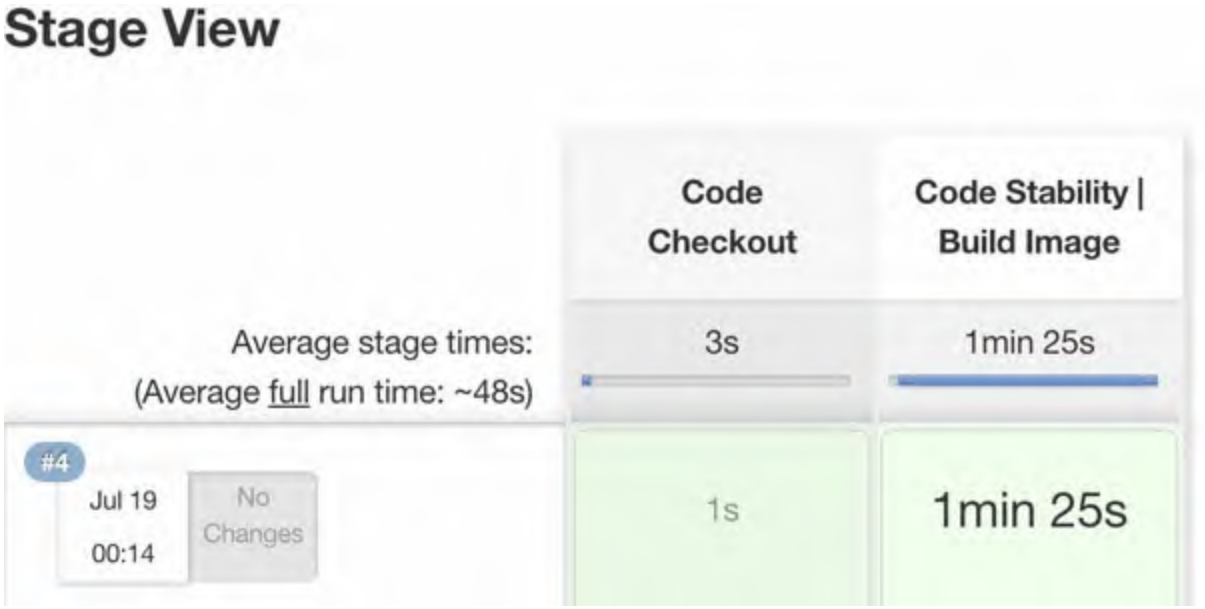

Figure 6.3: *Code Stability*

Code Quality

Once we have passed the stage for code stability, the next step is to check the quality of the code. Just like the previous legacy pipeline, we will use "**`Checkstyle`**" with Maven to check the quality of the code.

However, checking the code quality is insufficient since our application is now containerized using the Dockerfile. So, matching the quality of the Dockerfile also becomes very important from a standardization point of view. Now, when we say checking the quality of Dockerfile, it means that we are going to look at whether we are following the best practices for writing the Dockerfiles. For instance, some of the best practices for Dockerfiles are discussed here.

Ordering

It matters while writing a Dockerfile. If you see in the following example, removing **COPY** before **RUN** instructions will improve the image build time because of caching:

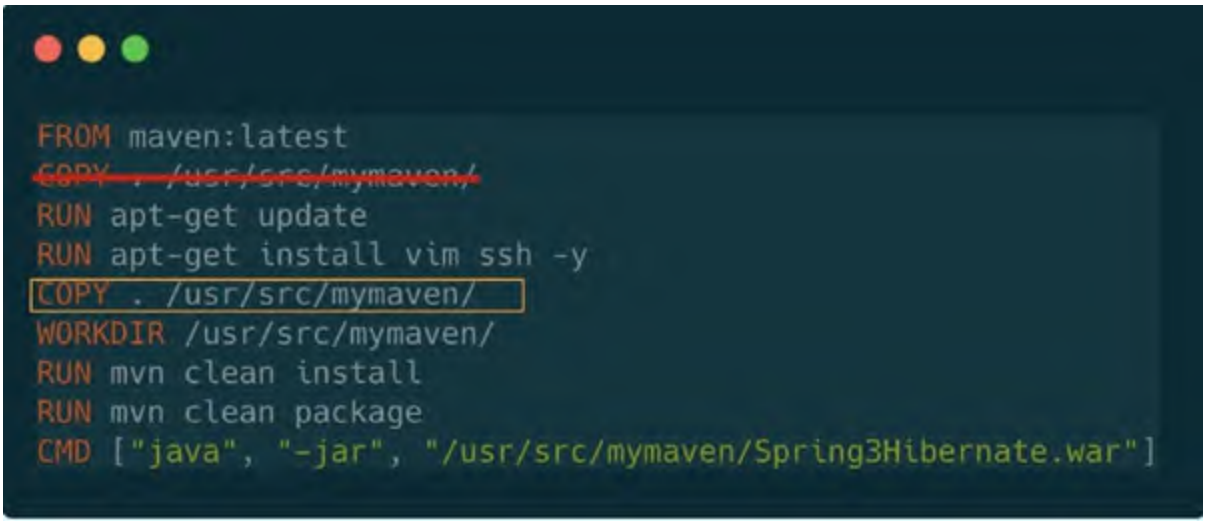

Figure 6.4: *Ordering*

Specific Files

Instead of copying the entire content of the directory, always copy the specific files because there could be some extra content, like- .git folder, README, and CHANGELOG, as shown in the following figure:

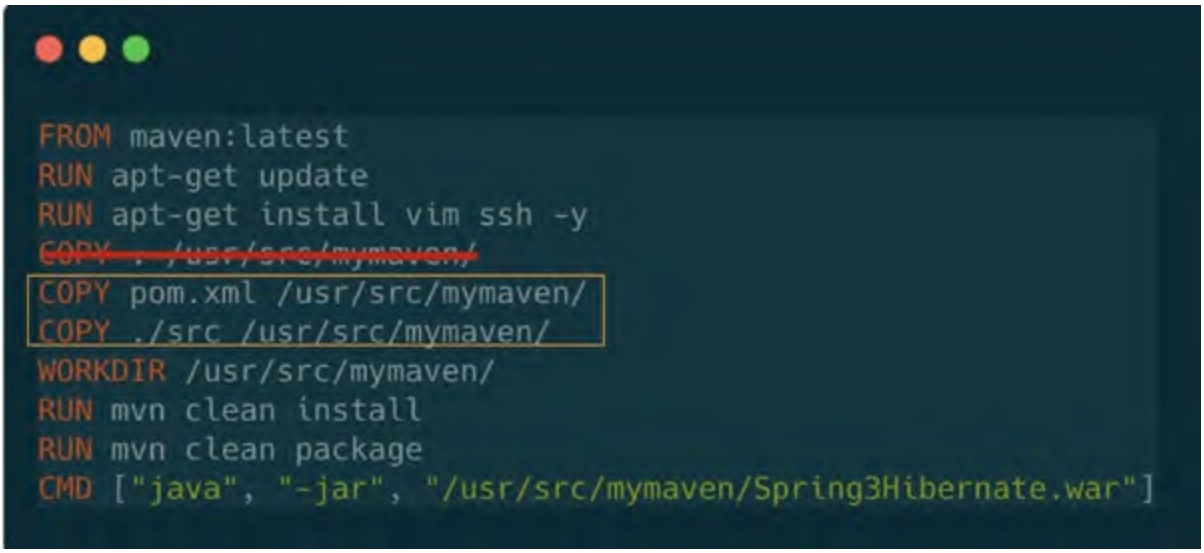

Figure 6.5: *Specific Files*

Multiline Instructions

As a best practice, we should always create a logical grouping on instruction, as shown in the following screenshot. We made a single **RUN** instruction to minimize the image layering here:

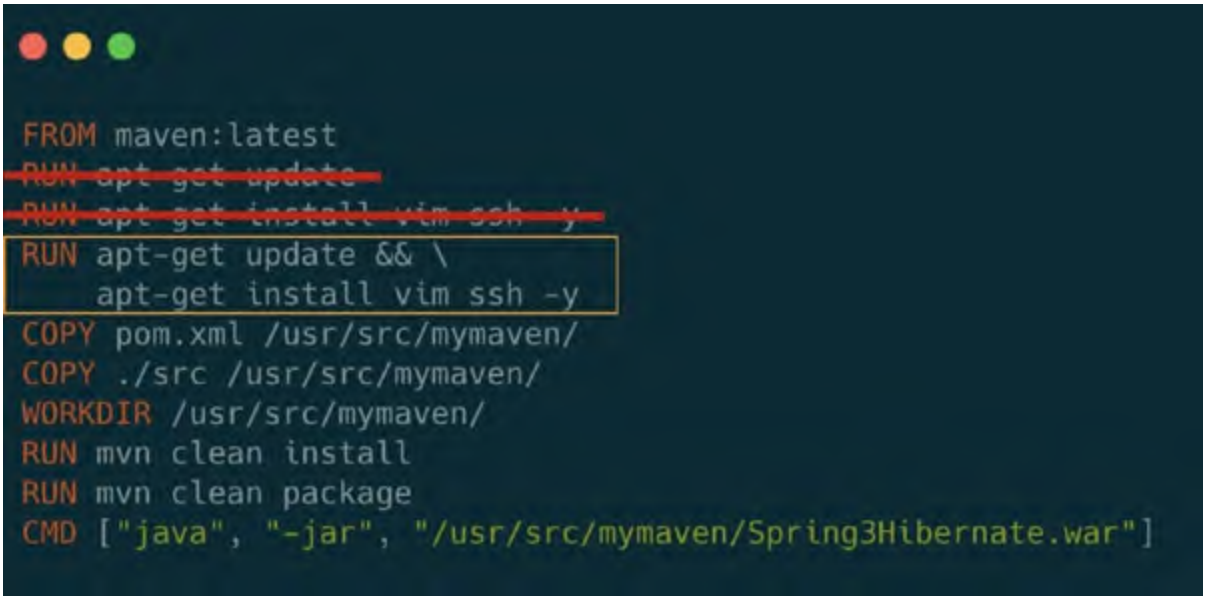

Figure 6.6: *Multiline Instructions*

Unnecessary packages

We should never install unnecessary packages inside our Docker image. For

debugging purposes, we can always install packages after the image is deployed but should never do it while building the images, as shown in the following figure we should not define packages before COPY statement:

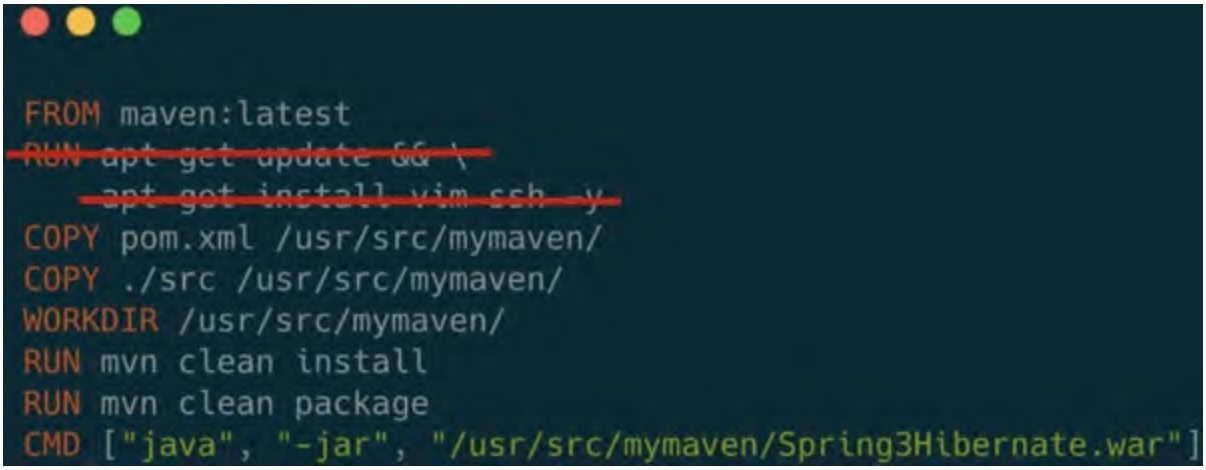

Figure 6.7: *Unnecessary Packages*

Package Manager Cache

Once all the requirements are installed, we should remove the package manager cache like apt-cache or yum cache, as shown in *Figure 6.8*:

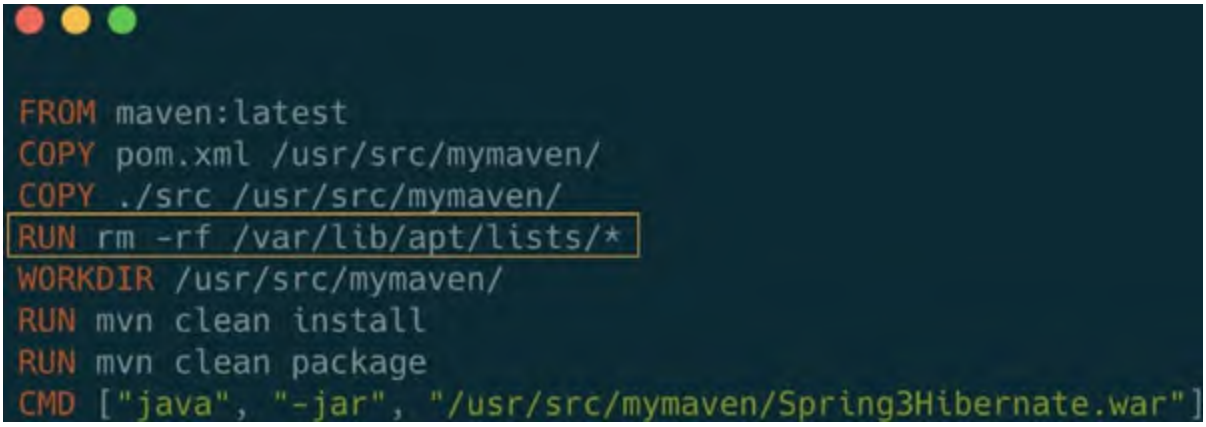

Figure 6.8: *Package Manager Cache*

Logical Grouping

`mvn clean install` needs to be executed when we are making a change in a pom.xml file so that only the upper layer will get changed if we are making changes in pom.xml. The codebase will get compiled without installing the dependencies again and again. This can be seen in the following figure:

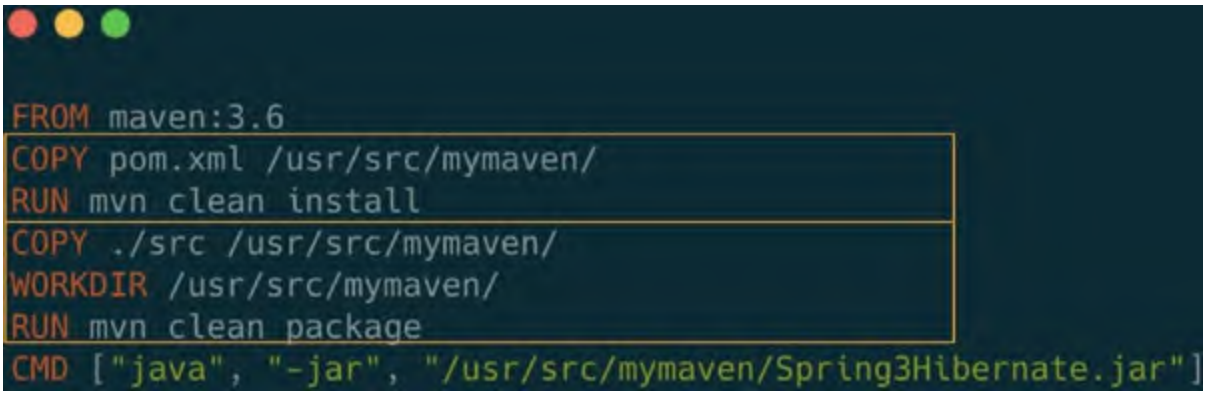

Figure 6.9: *Package Manager Cache*

Some other Dockerfile-related best practices are as follows:

- Using official images with specific tags instead of the latest tag
- Logical grouping of commands to minimize the layers
- Trying to use alpine flavor-based images to reduce the image size
- Using .dockerignore to ignore all irrelevant files in Docker build process
- Trying to use "Least Privileges User" in the Dockerfile to improve image security

We have listed a few best practices for Dockerfile, but other best practices are also available that we haven't shown in the examples above. There are many other checks and practices as well. But how can we ensure that we are always following the best practices?

Similar to Checkstyle, we have a tool named "**`hadolint`**" that can be used to test Dockerfiles for bugs and best practices. It has already been supporting the reporting and auditing format in Jenkins.

We only have to install hadolint on our Jenkins system, and it will be integrated into our pipeline.

```
$ wget
https://github.com/hadolint/hadolint/releases/download/v2.10.0/ha
Linux-x86_64
$ chmod +x hadolint-Linux-x86_64
$ sudo mv hadolint-Linux-x86_64 /usr/local/bin/hadolint
```

Code 6.4

The pipeline code for checking the "*Code Quality*" is as follows:

```
pipeline {
```

```
  agent any
  stages {
     stage(“Code Checkout”) {
     steps {
      git credentialsId: ‘git-creds’, url: ‘https://gitlab.com/
       ot-book/spring3hibernate.git’
      }
     }
     }
     stage(“Code Quality | Checkstyle | Hadolint”) {
     parallel {
      stage(“Checkstyle”) {
       agent {
        docker {
         args “-v ${HOME}/.m2:/root/.m2”
         image ‘opstreedevops/maven:java8’
        }
       }
       steps {
        git credentialsId: ‘git-creds’, url: ‘https://
        gitlab.com/ot-book/spring3hibernate.git’
        sh “mvn checkstyle:checkstyle”
        recordIssues(tools: [checkStyle(pattern: ‘**/
        checkstyle-result.xml’)])
       }
      }
      stage(“Hadolint”) {
       steps {
        sh “hadolint Dockerfile --no-fail -f json | tee -a
        hadolint.json”
        recordIssues(tools: [hadoLint(pattern: ‘hadolint.
        json’)])
       }
      }
     }
     }
  }
}
```

Code 6.5

Post execution, the stage view will look like this:

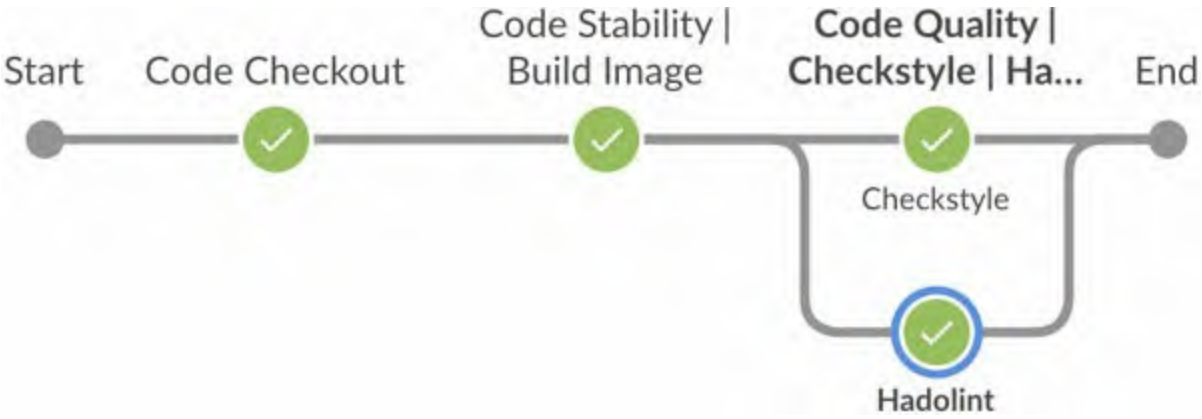

Figure 6.10: *Code Quality*

Hadolint and Checkstyle reports will be part of the Jenkins report publishing. We can view the reports from the Jenkins portal itself. The Checkstyle report will look like this:

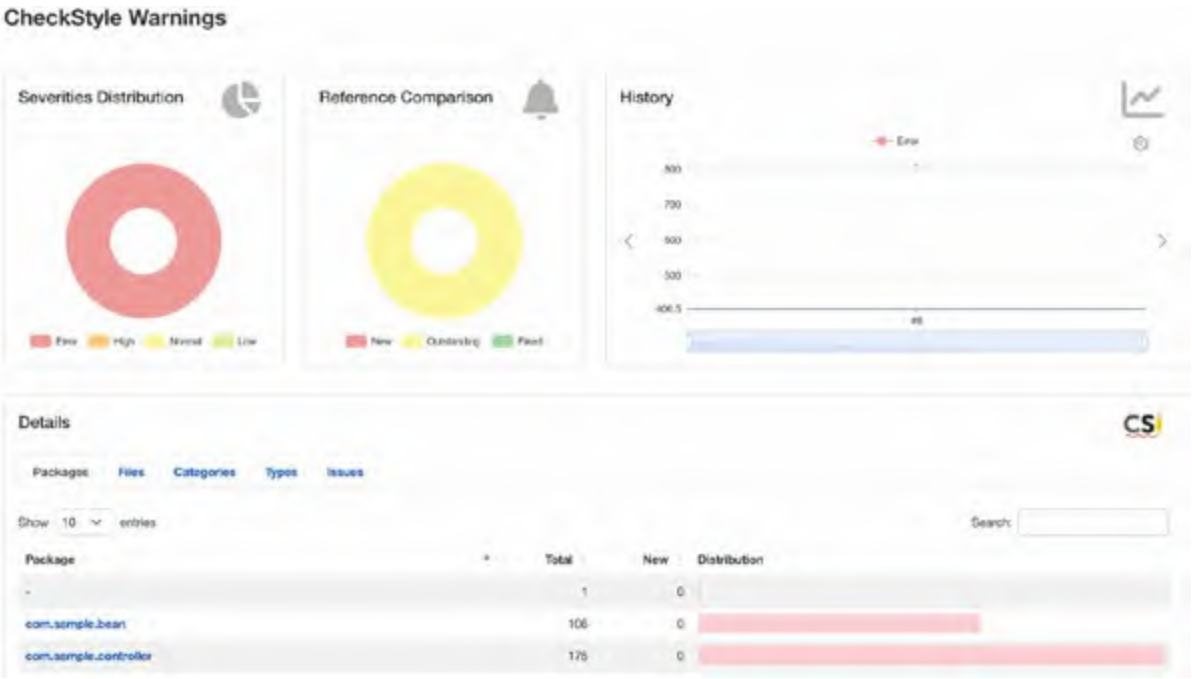

Figure 6.11: *Checkstyle Report*

The Hadolint report will look like the following:

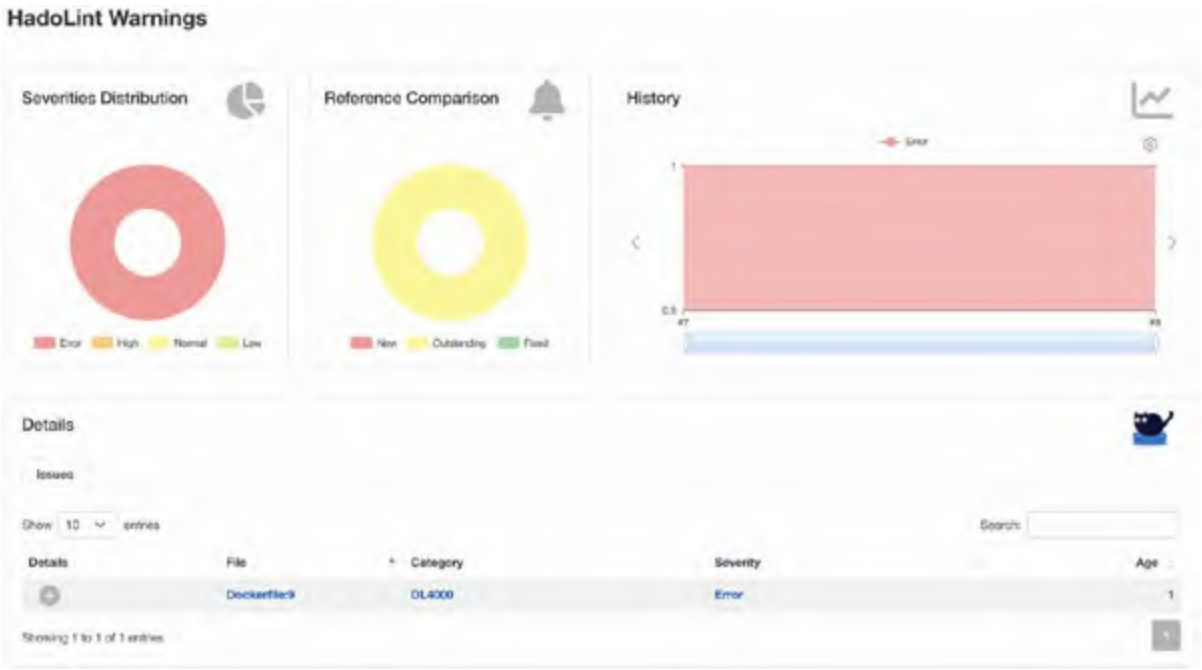

Figure 6.12: *Hadolint Report*

Unit Testing

As we know, “Test Cases” are already part of the system, and we have executed them in our legacy pipelines. Similarly, we will perform “Unit Testing” in this pipeline, but with a minor change, that is, the step will be executed inside a Docker container.

The pipeline view will look like this:

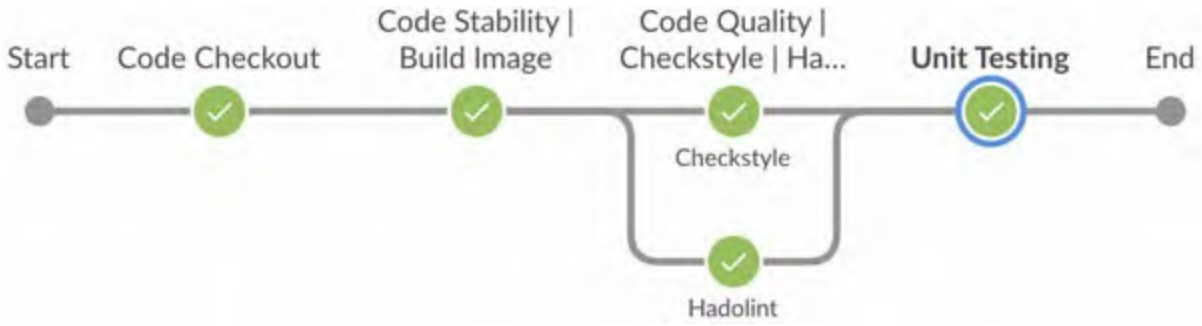

Figure 6.13: *Stage View | Unit Testing*

The pipeline code will be updated with Junit steps:

```
pipeline {
  agent any
  stages {
     stage(“Code Checkout”) {
```

```
  steps {
   git credentialsId: 'git-creds', url: 'https://gitlab.com/
    ot-book/spring3hibernate.git'
  }
  }
  stage("Code Stability | Build Image") {
  steps {
   script {
    docker.build("cloud.canister.io:5000/opstree/
       spring3hibernate:${env.BUILD_ID}")
   }
  }
  }
  stage("Code Quality | Checkstyle | Hadolint") {
  parallel {
   stage("Checkstyle") {
    agent {
     docker {
      args "-v ${HOME}/.m2:/root/.m2"
      image 'opstreedevops/maven:java8'
     }
    }
    steps {
     git credentialsId: 'git-creds', url: 'https://
      gitlab.com/ot-book/spring3hibernate.git'
     sh "mvn checkstyle:checkstyle"
     recordIssues(tools: [checkStyle(pattern: '**/
      checkstyle-result.xml')])
    }
   }
   stage("Hadolint") {
    steps {
     sh "hadolint Dockerfile --no-fail -f json | tee -a
      hadolint.json"
     recordIssues(tools: [hadoLint(pattern: 'hadolint.
      json')])
    }
}
  }
  }
  stage("Unit Testing") {
  agent {
   docker {
    args "-v ${HOME}/.m2:/root/.m2"
    image 'opstreedevops/maven:java8'
   }
```

```
    }
    steps {
      git credentialsId: 'git-creds', url: 'https://gitlab.com/
       ot-book/spring3hibernate.git'
      sh "mvn test"
      recordIssues(tools: [junitParser(pattern: 'target/
          surefire-reports/*.xml')])
    }
    }
  }
}
```

Code 6.6

The Junit report of the pipeline will be updated to the following after adding unit testing:

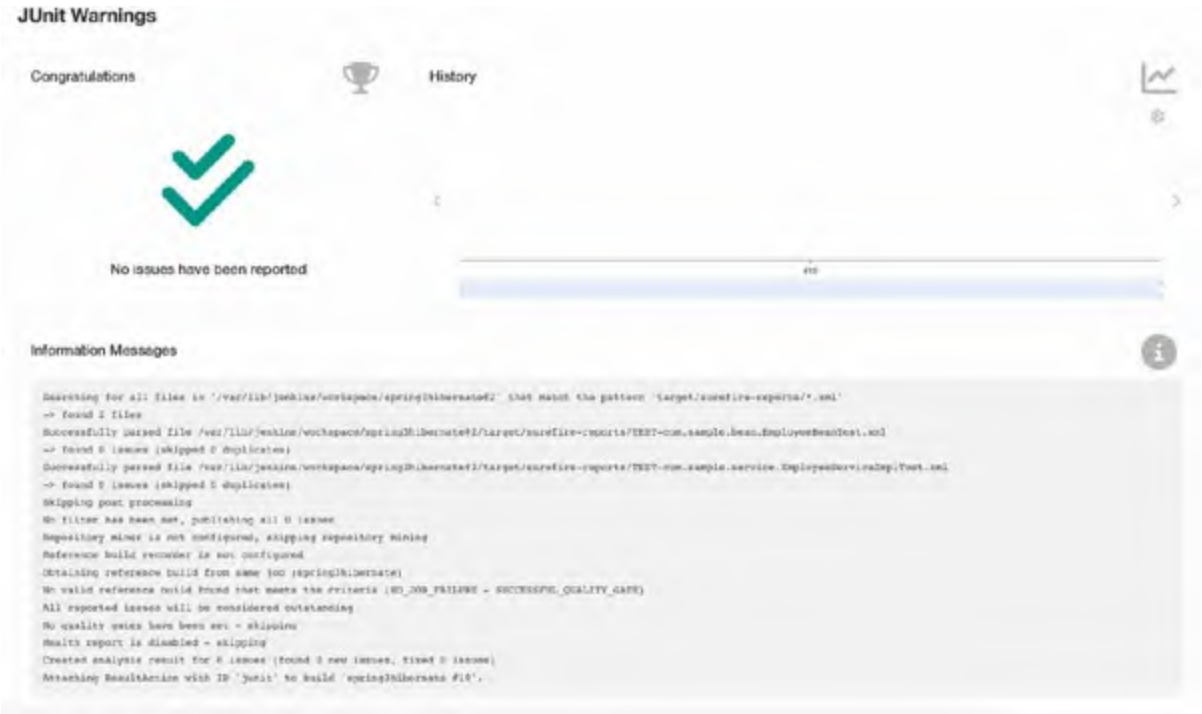

Figure 6.14: *Junit Report*

Code Coverage

Similar to Unit Testing, **Code Coverage** is also a very important aspect of application testing. We will be using the Cobertura for coverage testing. Again, this will be a part of a separate container step to avoid cluttering the main system.

Once we add the coverage testing inside the pipeline, the view will be updated like this:

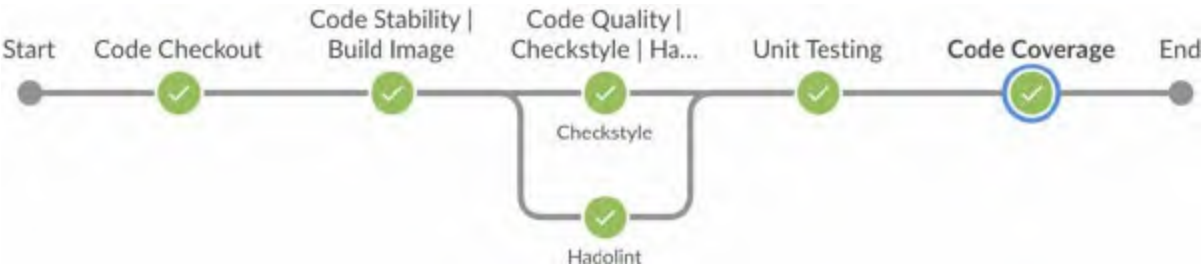

Figure 6.15: *Stage View | Code Coverage*

The stage changes will be done by the changes inside the pipeline code, and the updated pipeline code will look like this:

```
pipeline {
  agent any
  stages {
     stage("Code Checkout") {
     steps {
      git credentialsId: 'git-creds', url: 'https://gitlab.com/
       ot-book/spring3hibernate.git'
     }
     }
     stage("Code Stability | Build Image") {
     steps {
      script {
       docker.build("cloud.canister.io:5000/opstree/
        spring3hibernate:${env.BUILD_ID}")
      }
     }
     }
     stage("Code Quality | Checkstyle | Hadolint") {
     parallel {
      stage("Checkstyle") {
       agent {
        docker {
         args "-v ${HOME}/.m2:/root/.m2"
         image 'opstreedevops/maven:java8'
        }
       }
       steps {
        git credentialsId: 'git-creds', url: 'https://
         gitlab.com/ot-book/spring3hibernate.git'
        sh "mvn checkstyle:checkstyle"
        recordIssues(tools: [checkStyle(pattern: '**/
        checkstyle-result.xml')])
```

```
  }
 }
 stage(“Hadolint”) {
  steps {
   sh “hadolint Dockerfile --no-fail -f json | tee -a
   hadolint.json”
   recordIssues(tools: [hadoLint(pattern: ‘hadolint.
   json’)])
  }
 }
}
}
stage(“Unit Testing”) {
agent {
 docker {
  args “-v ${HOME}/.m2:/root/.m2”
  image ‘opstreedevops/maven:java8’
 }
}
steps {
 git credentialsId: ‘git-creds’, url: ‘https://gitlab.com/
  ot-book/spring3hibernate.git’
 sh “mvn test”
 recordIssues(tools: [junitParser(pattern: ‘target/
  surefire-reports/*.xml’)])
}
}
stage(“Code Coverage”) {
agent {
 docker {
  args “-v ${HOME}/.m2:/root/.m2”
  image ‘opstreedevops/maven:java8’
 }
}
steps {
 git credentialsId: ‘git-creds’, url: ‘https://gitlab.com/
  ot-book/spring3hibernate.git’
 sh “mvn cobertura:cobertura”
 cobertura autoUpdateHealth: false, autoUpdateStability:
 false,
   coberturaReportFile: ‘**/target/site/cobertura/
   coverage.xml’,
   conditionalCoverageTargets: ‘70, 0, 0’,
   failUnhealthy: false,
   failUnstable: false, lineCoverageTargets: ‘80, 0, 0’,
   maxNumberOfBuilds: 0, methodCoverageTargets: ‘80, 0, 0’,
   onlyStable: false, sourceEncoding: ‘ASCII’,
   zoomCoverageChart: false
```

```
        }
        }
    }
}
```

Code 6.7

After adding Code Coverage, the report can be visualized over Jenkins:

Figure 6.16: *Cobertura Report*

Security Testing

In our legacy pipeline, we added the “Security Testing” stage. At that stage, we scanned the bugs and vulnerabilities in our application using a popular **Static Application Security Testing** (**SAST**) tool, **OWASP**.

OWASP dependency check is a capable and very powerful tool in terms of application vulnerability testing. But since we are packaging our application in container form, there might be some issues/bugs/vulnerabilities inside the container as well that can increase the potential attack surface for our application. In such scenarios, we must assess our container images as well for low, moderate, and high-risk issues/vulnerabilities.

Another important reason behind performing this assessment is that we

mostly inherit public images to build our application images, and there might be some misconfiguration and vulnerabilities inside public images; so, we must take care to scan and analyze dependencies and packages inside a container image.

For the checklist, the **Center for Internet Security** (**CIS**) publishes the dependencies and packages that have vulnerabilities and explains how we can resolve such vulnerabilities. However, the tough part is automating these checks and scanning the pipeline so that we do not miss any big security risks. For these kinds of scenarios, the open-source and enterprise marketplace released different software under the category named "*Image Scanning Tools*".

These image-scanning tools can extract the container images layer by layer and scan every dependency or package available in the system. Based on the analysis, they provide reports with low, moderate, and high risks. Some of the most popular image scanning tools are listed here:

- **Trivy: https://github.com/aquasecurity/trivy**
- **Anchor: https://anchore.com/**
- **Clair: https://github.com/quay/clair**
- **AquaScanner: https://www.aquasec.com/**
- **Snyk: https://snyk.io/**

All these tools use the **Common Vulnerabilities and Exposures** (**CVE**) databases and CIS benchmarks to compare vulnerabilities, but the implementation and reporting mechanism is different for each. After some investigation, we finalized that we would go ahead with Snyk as a security testing tool for the container. One of the primary reasons for using Snyk is that it can scan like OWASP so that a single tool can serve the purpose for us.

Here's a little bit about Snyk:

"*It is a tool that can scan and analyze application codebase like Java, Python, Golang, and so on. Other than application scanning, it can be used for identification of open-source dependencies, container image vulnerabilities, and* ***Infrastructure as Code (IaC)*** *tools like terraform.*"

To integrate Snyk with Jenkins, we simply must install the Snyk plugin inside it, as shown in the following figure:

Figure 6.17: *Snyk Plugin*

Once the plugin is installed, we have to do the following configurations inside the "*Global Tool Configuration*":

Figure 6.18: *Snyk Tool*

As the last part of Snyk integration, we need to create a Jenkins credential for the Snyk token, as shown in the following figure:

Figure 6.19: *Snyk Token*

Once these changes have been made to the Jenkins system, we can start security testing in our pipeline. The code snippet will be like this after adding the security testing:

```
pipeline {
  agent any
  stages {
     stage("Code Checkout") {
     steps {
      git credentialsId: 'git-creds', url: 'https://gitlab.com/
       ot-book/spring3hibernate.git'
     }
     }
     stage("Code Stability | Build Image") {
     steps {
      script {
       docker.build("cloud.canister.io:5000/opstree/
       spring3hibernate:${env.BUILD_ID}")
      }
     }
     }
     stage("Code Quality | Checkstyle | Hadolint") {
     parallel {
      stage("Checkstyle") {
       agent {
        docker {
         args "-v ${HOME}/.m2:/root/.m2"
```

```
      image 'opstreedevops/maven:java8'
     }
    }
    steps {
     git credentialsId: 'git-creds', url: 'https://
     gitlab.com/ot-book/spring3hibernate.git'
     sh "mvn checkstyle:checkstyle"
     recordIssues(tools: [checkStyle(pattern: '**/
     checkstyle-result.xml')])
    }
  }
  stage("Hadolint") {
    steps {
     sh "hadolint Dockerfile --no-fail -f json | tee -a
     hadolint.json"
     recordIssues(tools: [hadoLint(pattern: 'hadolint.
     json')])
    }
  }
}
}
stage("Unit Testing") {
agent {
  docker {
    args "-v ${HOME}/.m2:/root/.m2"
    image 'opstreedevops/maven:java8'
  }
}
steps {
  git credentialsId: 'git-creds', url: 'https://gitlab.com/
    ot-book/spring3hibernate.git'
  sh "mvn test"
  recordIssues(tools: [junitParser(pattern: 'target/
    surefire-reports/*.xml')])
}
}
stage("Code Coverage") {
agent {
  docker {
    args "-v ${HOME}/.m2:/root/.m2"
    image 'opstreedevops/maven:java8'
  }
}
steps {
  git credentialsId: 'git-creds', url: 'https://gitlab.com/
    ot-book/spring3hibernate.git'
  sh "mvn cobertura:cobertura"
  cobertura autoUpdateHealth: false, autoUpdateStability:
```

```
      false,
       coberturaReportFile: '**/target/site/cobertura/
        coverage.xml',
       conditionalCoverageTargets: '70, 0, 0',
        failUnhealthy: false,
       failUnstable: false, lineCoverageTargets: '80, 0, 0',
       maxNumberOfBuilds: 0, methodCoverageTargets: '80, 0, 0',
       onlyStable: false, sourceEncoding: 'ASCII',
        zoomCoverageChart: false
    }
    }
    stage("Security Testing | OWASP | Snyk") {
    parallel {
     stage("Dependency Scan") {
      agent {
       docker {
        args "-v ${HOME}/.m2:/root/.m2"
        image 'opstreedevops/maven:java8'
       }
      }
      steps {
       git credentialsId: 'git-creds', url: 'https://
        gitlab.com/ot-book/spring3hibernate.git'
       sh "mvn org.owasp:dependency-check-maven:check"
       publishHTML([allowMissing: false, alwaysLinkToLastBuild:
       true, keepAll: true, reportDir: 'target', reportFiles:
       'dependency-check-report.html', reportName: 'Dependency
       Check', reportTitles: 'Spring3hibernate'])
      }
     }
     stage("Container Scan") {
      steps {
       snykSecurity snykInstallation: 'snyk', snykTokenId:
       'snyk', additionalArguments: "--docker
       cloud.canister.io:5000/opstree/spring3hibernate:${env.BUI
       failOnError: false
      }
     }
    }
    }
  }
}
```

Code 6.8

After adding the stages for security testing, the view of pipeline would be modified like this:

Figure 6.20: *Stage View | Security Testing*

In *figure 6.21*, report for OWASP Dependency check is shown:

Figure 6.21: *Dependency check report*

In *figure 6.22*, report for Container Security Scan is shown:

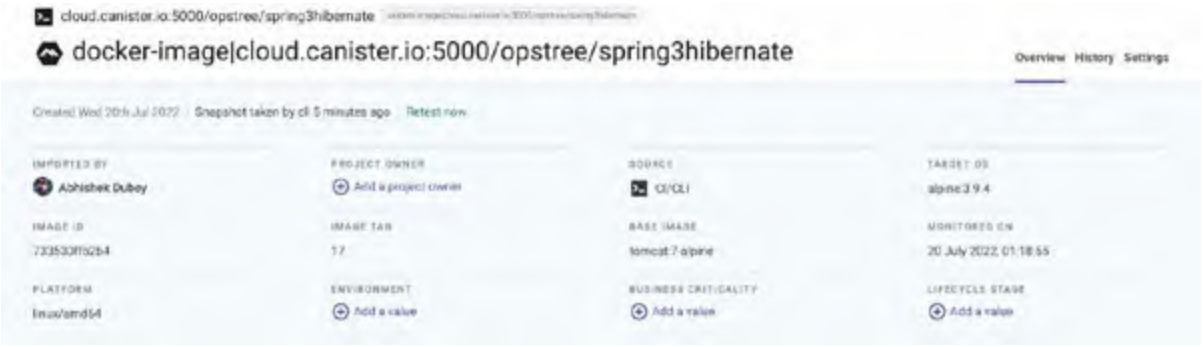

Figure 6.22: *Container Report*

Another example of Container report can be seen in the following figure:

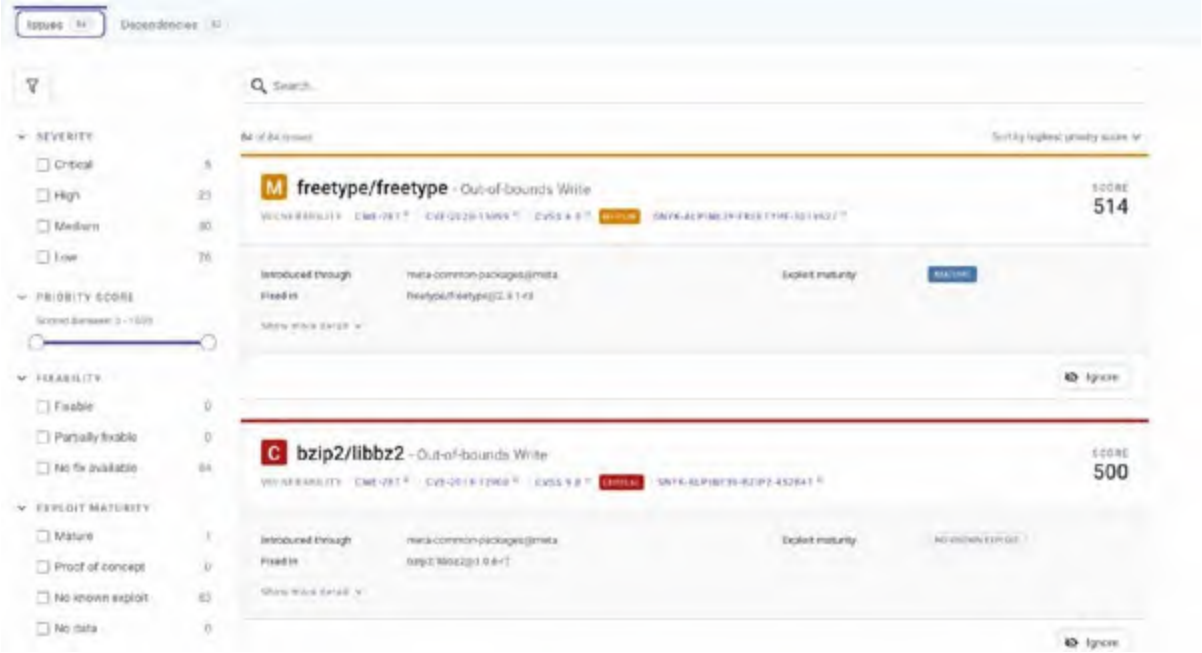

Figure 6.23: *Container Report*

Conclusion

Not so complicated, huh? Anything can be made easy when broken down into steps. This is not to say that all this is foolproof or that implementation is often quite simple. Things change at the drop of a hat: new requirements come in, manual errors are made, tool versions are updated, services are onboarded with different prerequisites, and so on. The important thing to remember is that we need to cover the basic steps in our CI pipeline to ensure proper standardization and hit security benchmarks. We need to be especially sure of the latter because we do not want to fall into the already identified vulnerabilities trap. It will not only look bad but can also compromise our services/data and cause serious damage. All the other case-specific problems can be dealt with through good old engineering work: planning, designing, implementation, and troubleshooting.

In the next chapter, we will discuss CD. Technically, it comes after CI, and we're going to stay on that path. We'll be talking about what problems CD solves, CD testing elements, deployment strategies, and much more.

CHAPTER 7

Continuous Deployment

After completing the sprint, the team was very happy and full of positive energy because they had achieved a massive milestone of running a smooth Continuous Integration pipeline to increase development productivity. There was a significantly lower number of bugs and issues in the production environment, causing downtime. In addition to Continuous Integration, Docker is \introduced to the system to help maintain code sanity across the environments. The application was successfully dockerized, and the CI steps were integrated with Docker. Now, the difference can be easily seen in the form of a bugs/issues graph, as shown in the following figure:

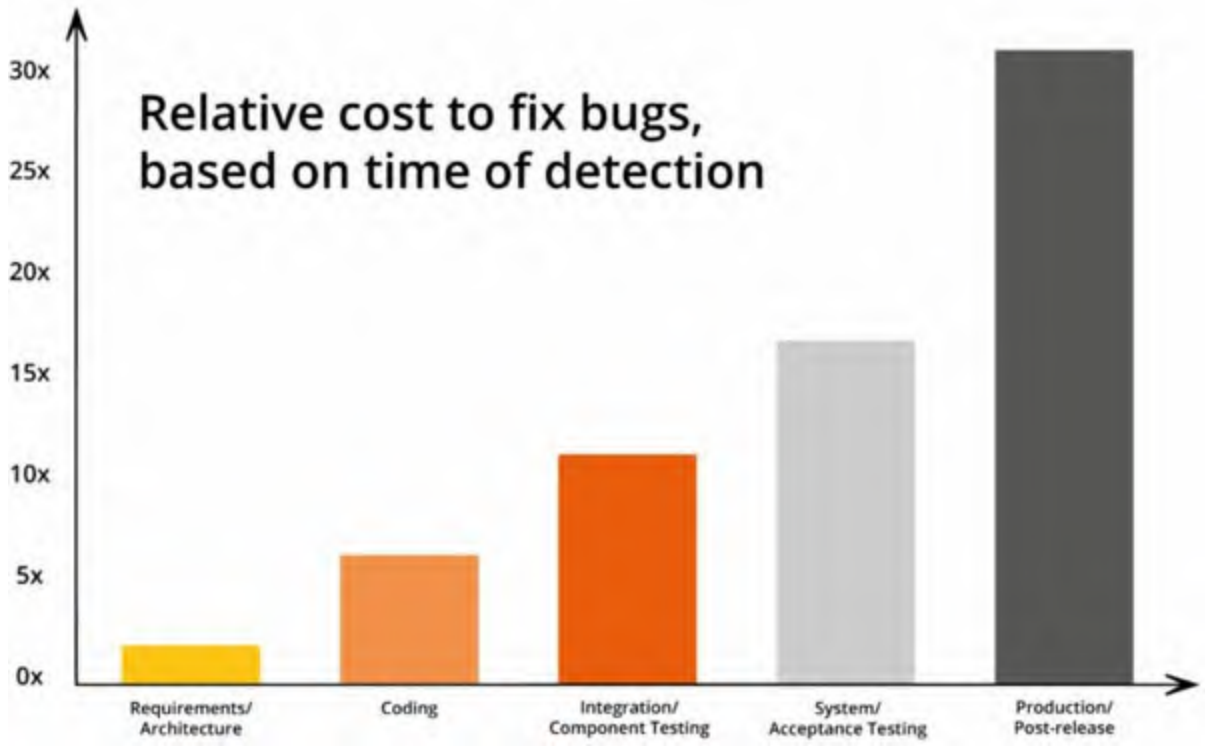

Figure 7.1: *Production issues graph form*

As humans, we have a nature of thinking about the future, so now the team is concerned about the deployment of different environments like QA, security,

performance, and business. Since, as of now, the deployment of code is happening manually in different environments, sometimes issues are being created like steps missing, deployment steps inconsistency, and rollback challenges.

Abhishek was responsible for exploring all Continuous Deployment testing and deployment strategies.

Structure

In this chapter, we will discuss the following topics:

- Different kinds of environments
- CD Testing elements
- Deployment strategies

Objectives

This chapter will help you understand the concepts of different environments in the software development cycle. Also, this chapter talks about the purpose and importance of each environment. It will also discuss different deployment strategies like Normal, Rolling Update, Blue Green, and Canary.

The pros and cons of each deployment strategy will be discussed, and we will look at how we can choose the ideal deployment strategy for different scenarios.

Different Kinds of Environments

Abhishek thought that deployment in the development environment was successful, but it will be an excellent decision to deploy the application in a Production environment because the deployment was successful in the Development environment. Also, if there are going to be different environments, what types of environments will they be?

After reading many blogs, journals, and documentation, he was amazed at how people utilize multiple environments for different purposes.

QA environment

Quality Assurance (**QA**) environment is an environment that mimics the production environment. A QA environment is generally used by QA engineers, analysts, or other testing professionals to perform functional and non-functional testing of services.

It is a kind of safety net to catch all the unwanted bugs and issues before release. Ideally, the QA environment is a different environment in which QA engineers perform testing from the QA and the user's perspective. The QA environment aims to ensure reliability and confidence in the application codebase.

QA environment plays a vital role in software development and releases because, in this environment, rigorous system testing is done to identify any leaks or bugs that can cause harm to the software and the company's reputation. Generally, QA engineers perform different types of testing:

- Regression testing
- BDD testing
- Security testing
- API testing
- Performance testing

Security testing environment

In the QA environment, testing is primarily focused on the functional and non-functional parts of the application. But when we talk about software testing and release, security becomes a priority because ignorance of security can lead to complete system hacks and shutdowns.

To identify all types of security issues and gaps, we set up a separate environment called the "*Security Testing Environment*." Just like our soldiers prepare for any kind of war coming their way, the intent of the security testing environment is to simulate security attacks on the software and application to identify security-related bugs and issues.

As we already know, **Static analysis security testing** (**SAST**) gets performed in the CI pipeline. There are some security measures that we cover after application deployment. Those testing methods are generally called **Dynamic analysis security testing** (**DAST**). Some good DAST analysis tools are listed below:

- **Zed Attack Proxy: https://www.zaproxy.org/**
- **Nikto: https://github.com/sullo/nikto**
- **Accunetix: https://www.acunetix.com/**
- **AppScan: https://cloud.appscan.com/**

For security testing, organizations generally set up a separate environment so that the security testing would not mess up the existing QA testing environment.

Performance Testing Environment

A performance testing environment is an evaluation environment where the performance of applications and infrastructure is evaluated. It's a process of assessing the quality and its functioning when the software is released. Performance testing gets conducted to check system stability and responsiveness.

Load/stress testing is one of the simplest performance testing methods on applications. There are various performance testing tools available. Some of those are mentioned here:

- **Load ninja: https://loadninja.com/**
- **Apache Jmeter: https://jmeter.apache.org/**
- **Loadrunner: https://en.wikipedia.org/wiki/LoadRunner**
- **Locust: https://locust.io/**

Performance testing environment is generally dynamic, which means it is not up 24*7. The teams create the performance testing environment separately to test the application and infrastructure performance, and once testing is completed, they switch off the environment. This is the primary reason for keeping the performance testing environment separate from the other testing environments. Another reason is that performance testing can hamper the existing testing environment, so it always makes sense to have a separate environment for performance testing.

Business Testing Environment

User Acceptance Testing (**UAT**) environment is a testing environment in which the software is tested from an end-user perspective in the real world by

the intended consumers. This is the last software testing phase before releasing a software/application.

UAT effectively ensures quality in terms of cost while increasing transparency for consumers. It helps the developers work with real issues and data. The software is released into the production environment if this testing phase is completed successfully.

There are multiple testing phases inside the UAT environment:

- Alpha testing
- Beta testing

UAT environment is always set up as a separate environment and is most close to the production environment. Even the infrastructure would be identical to the production environment so that there are zero discrepancies while testing applications with real users or consumers.

CD Testing Elements

Abhishek was happy with his analysis of different environments, but there were a few more things that needed to be cleared out; a major thing was the CD testing elements. So, Abhishek approached Sonia because she has a good understanding of and experience with the QA testing environment.

He asked Sonia what testing and tools needed to be part of the continuous testing environment. She helped him by providing the details about testing phases in continuous deployment:

- Regression testing
- BDD testing
- Security testing
- API testing
- Performance testing

But she wasn’t sure about the tools that can be used for testing in these phases. She asked Abhishek to evaluate the tools related to the phases. And as we know, Abhishek is a seeker of knowledge; he started reading about these phases and the related tools.

Regression Testing

In regression testing, we rerun the automated tests to check the previously working and new functionalities. Typically, we do regression testing in these circumstances:

- A new feature is introduced
- Some of the bugs are fixed
- Performance improvements
- Configuration-level changes

While functional tests inspect the behavior of new changes or features, they don't check how much they are compatible with existing ones. Therefore, it would be challenging and time-consuming to identify the root cause of product failure without regression testing.

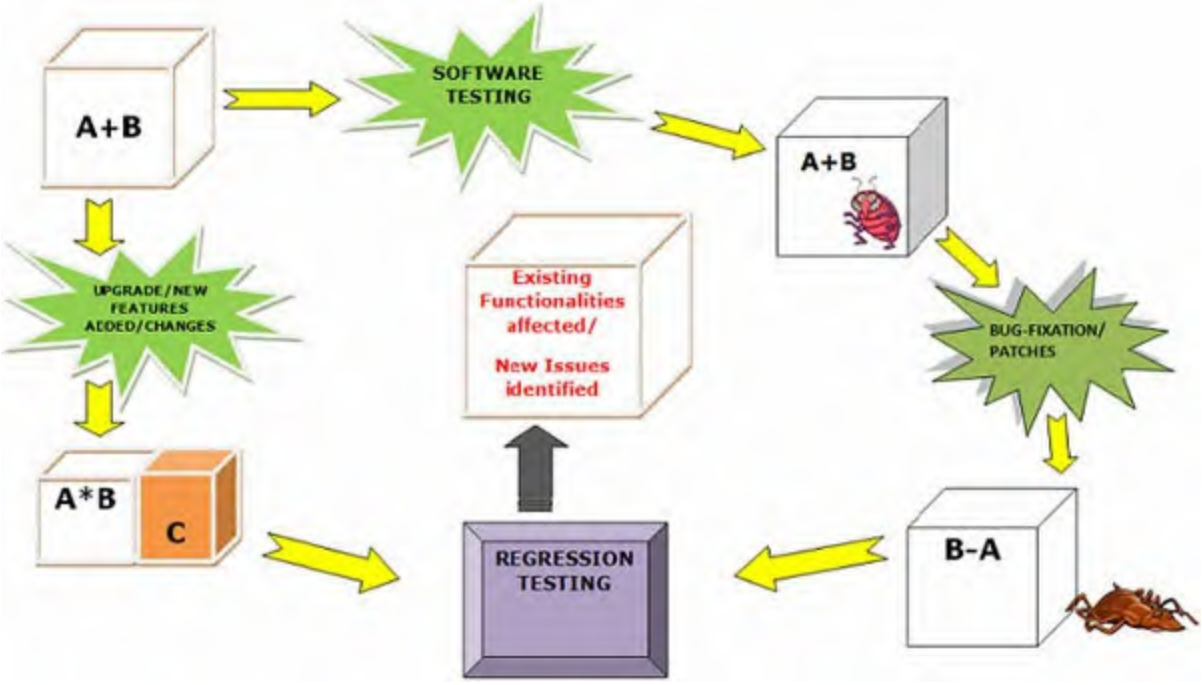

Figure 7.2: *Regression testing*

There are multiple tools available in the market to perform regression testing:

- **Selenium: https://www.selenium.dev/**
- **AppSurify: https://appsurify.com/**
- **Webking: https://www.embeddedtechnology.com/doc/webking-0001**

- **Test drive: https://origsoft.com/browser-based-automated-testing-tools/**

After a comparison between a lot of commercial and open-source software for regression testing, Abhishek and QA team finalized Selenium as the tool for regression testing after the deployment because of its reliability, ease, and excellent community support.

Selenium

Selenium is one of the foremost renowned open-source test automation frameworks. It allows test automation of web applications or websites across different browsers and operating systems. It offers compatibility with multiple programming languages like Java, JavaScript, Python, C#, and more, allowing testers to automate their website testing in any programming language they're comfortable with.

Selenium works with different browsers in headless or non-headless mode. A non-headless browser has a GUI portal available, for example, Chrome, Firefox, Internet Explorer, and so on. A headless browser is a web browser without any GUI. We need it because our application will open on different browsers, and we must ensure that it runs perfectly on every browser. The reason behind keeping it headless is that we are doing automated testing, and GUI will become a hindrance in that. The Selenium architecture can be seen in the following figure:

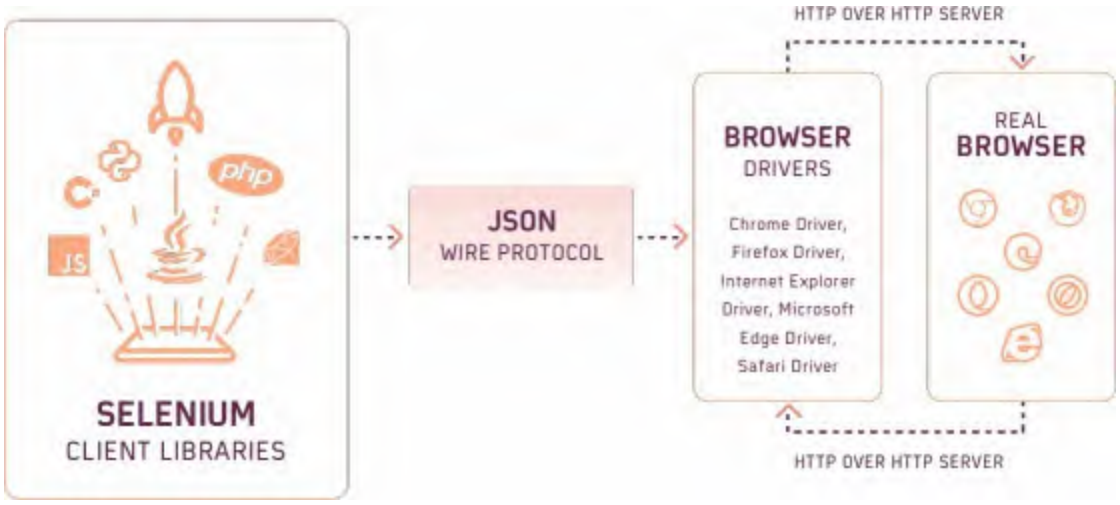

Figure 7.3: *Selenium Architecture*

Behavior Driven Development testing

Behavior Driven Development (**BDD**) testing is a technique of testing that validates the application behavior. It is an extension of **Test Driven Development (TDD)**. Generally, behavior-driven test cases are written in everyday language (mostly English) so that a non-technical person can also understand them.

BDD test cases get written in the Given-When-Then condition:

- Given (some context)
- When (something happens)
- Then (outcome/results)

A simple example could be as follows:

- Given: I am signing up for a free trial
- When: I provide correct information
- Then: My account is created, and I received the link to download

The BDD Architecture can be seen in the following figure:

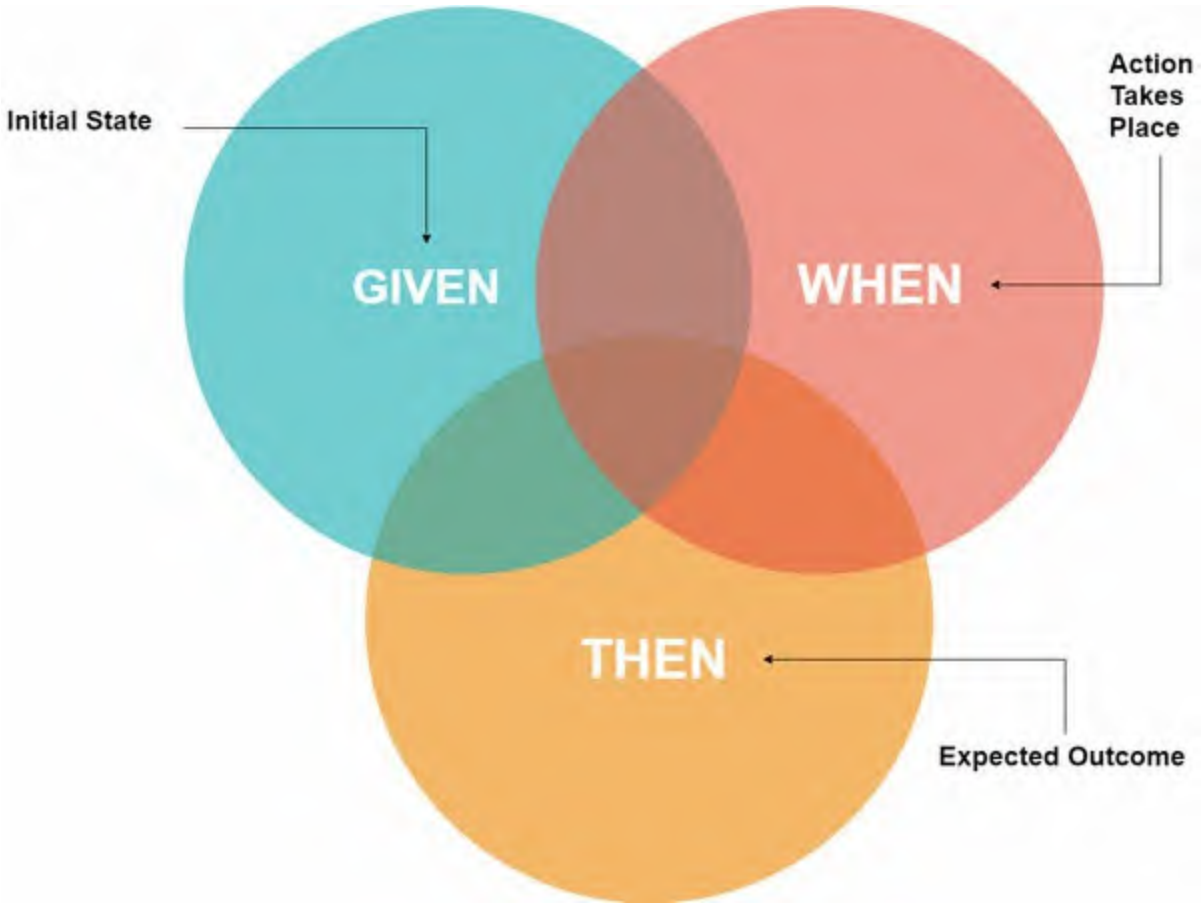

Figure 7.4: *BDD Architecture*

After some investigation, Abhishek stuck to the BDD testing tool and asked Sajal for help regarding this. Sajal had contacts from his previous organization and asked them for suggestions related to the BDD tool. One of Sajal's colleagues from the old organization asked him to evaluate Cucumber, which was suggested to Abhishek.

Abhishek evaluated Cucumber and found it a perfect match for the use case because it provides the capability of writing the BDD test cases without in-depth knowledge of programming languages. Also, it has excellent community support.

Cucumber

Cucumber is an automation framework for BDD testing. It executes the acceptance test cases using the Gherkin language (Given-When-The).

In the following figure, we can see an example of a Cucumber file:

```
# Comment
@tag
Feature: Eating too many cucumbers may not be good for you

  Eating too much of anything may not be good for you.

  Scenario: Eating a few is no problem
    Given Alice is hungry
    When she eats 3 cucumbers
    Then she will be full
```

Figure 7.5: *Cucumber File*

Cucumber architecture looks like this:

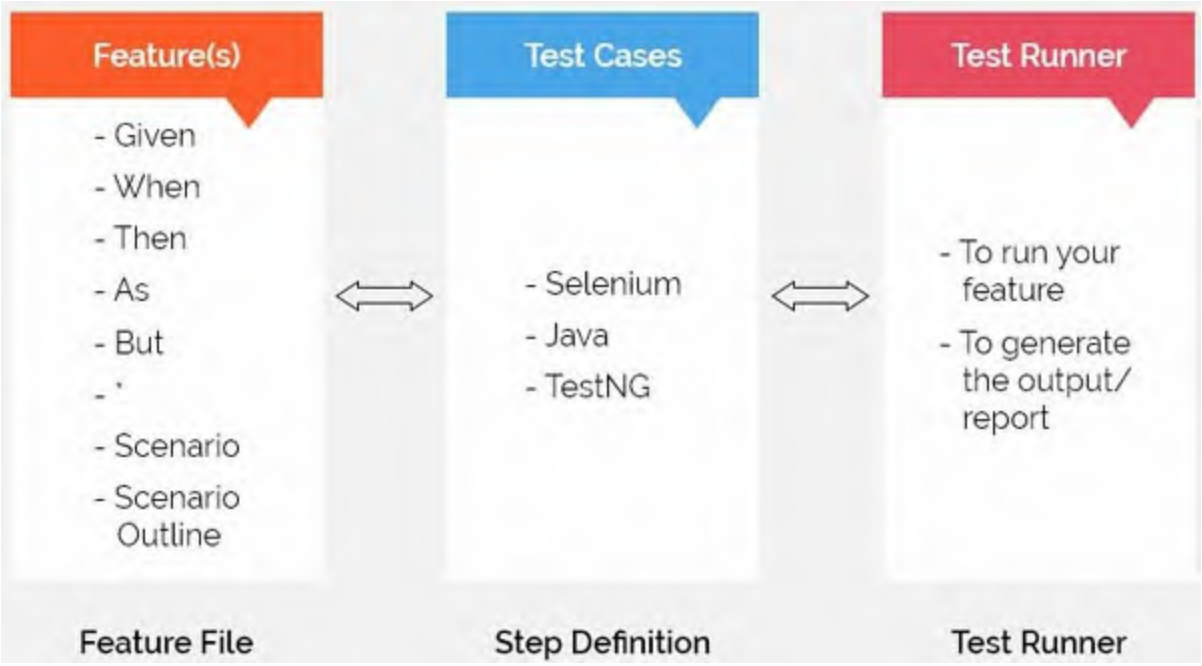

Figure 7.6: *Cucumber Architecture*

A feature file is a simple text file in which we write our test conditions in the Gherkin language, but it requires a test case framework as well to integrate with. To create these test case files, minimal coding will be required. The last part of the architecture is the Test runner, where these will be executed.

Security Testing

Security testing is a testing type used to identify security-related issues, vulnerabilities, and risks in the software. The aim of security testing is to prevent any malicious attack on the application because it can put the

application at full risk.

It focuses on identifying the software's vulnerabilities and loopholes before the attacker finds out and steals the valuable information.

Security testing can be divided into two categories:

- **SAST: Static Application Security Testing** (**SAST)** is a way of scanning the application code for any kind of vulnerabilities and possible loopholes. The code is checked extensively for any vulnerable dependencies and logic that can cause security breaches like XSS attacks and SQL injection.
- **DAST: Dynamic Application Security Testing** (**DAST**) is a BlackBox testing in which the attacks are simulated on running applications without knowing anything about the source code. This testing is generally performed with a hacker's mindset, to identify any kind of vulnerabilities that a real hacker can exploit.

In the deployment cycle, the security testing team performs DAST analysis on the applications. Abhishek had a friend who worked in government security and asked what tool could be used for DAST analysis. He recommended OWASP **Zed Attack Proxy** (**ZAP**) to be used for this testing. It's open source, easy to set up, and offers good documentation.

OWASP ZAP

OWASP ZAP is an open-source tool to analyze security issues and vulnerabilities while the application is deployed and running. ZAP has multiple modes according to a person's capability. For example, it can be used by beginners as well as experts.

It also supports different types of scanning mechanisms, like **Active and Passive** scans. One thing that sets ZAP apart from other web application security testing tools is its ability to be automated. However, it is still frequently used by penetration testers or individuals running manual security tests.

Figure 7.7 shows the ZAP architecture:

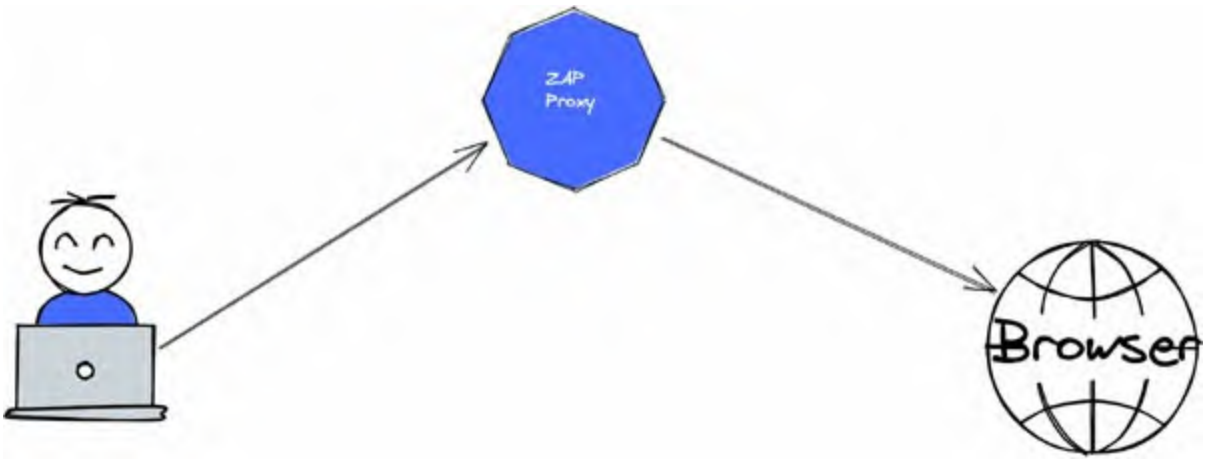

Figure 7.7: *ZAP Architecture*

The following figure shows a ZAP Scan result:

Figure 7.8: *ZAP Scan result*

API Testing

It's only sometimes the case that only the frontend application is failing, the frontend application can display the error, but it might be coming from the back-end application (generally, the API). The team needs to test the API for its functionality and responsiveness in such scenarios.

API testing includes its testing in terms of responsiveness, security, functionality, and reliability. It validates the logic of different endpoints exposed by the API.

A few things that are covered in API testing are as follows:

- JSON validation
- Header validation
- Request method validation
- Authentication and authorization validation

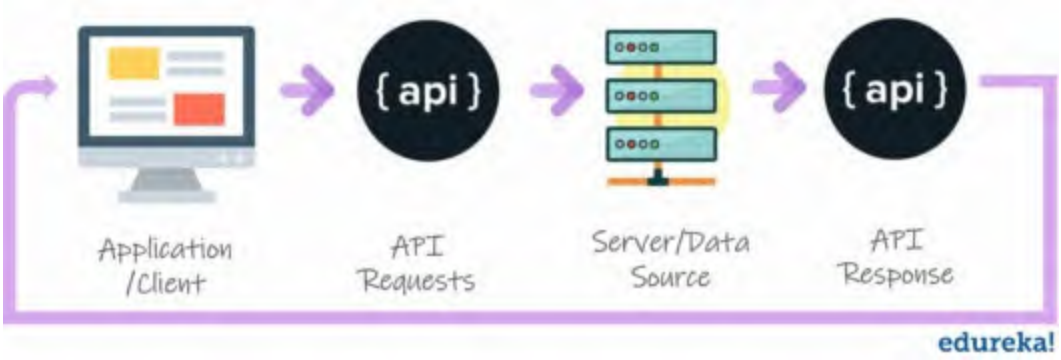

Figure 7.9: *API testing Architecture*

Abhishek did complete research on his own to identify how they can integrate API testing in their CD pipeline and came up with a solution of using SOAP UI with their automation. It is an industry standard for doing API testing on different platforms. Also, it can be integrated with multiple automation tools, like Jenkins and Gitlab CI.

Simple Object Access Protocol UI

Simple Object Access Protocol (**SOAP**) is an open-source tool to test the standard and measures of different APIs. It works with all kinds of APIs and their messaging protocols. For example, it supports JSON, XML, Javascript, and Text payload that can be sent to API.

It provides a simple interface for testing that can be used by both technical and non-technical people. Additionally, it can be used for security testing where headers and CORS policy can be validated. Many technical people use the SOAP tool to perform load testing on their APIs to check their reliability and performance, as shown in the following figure:

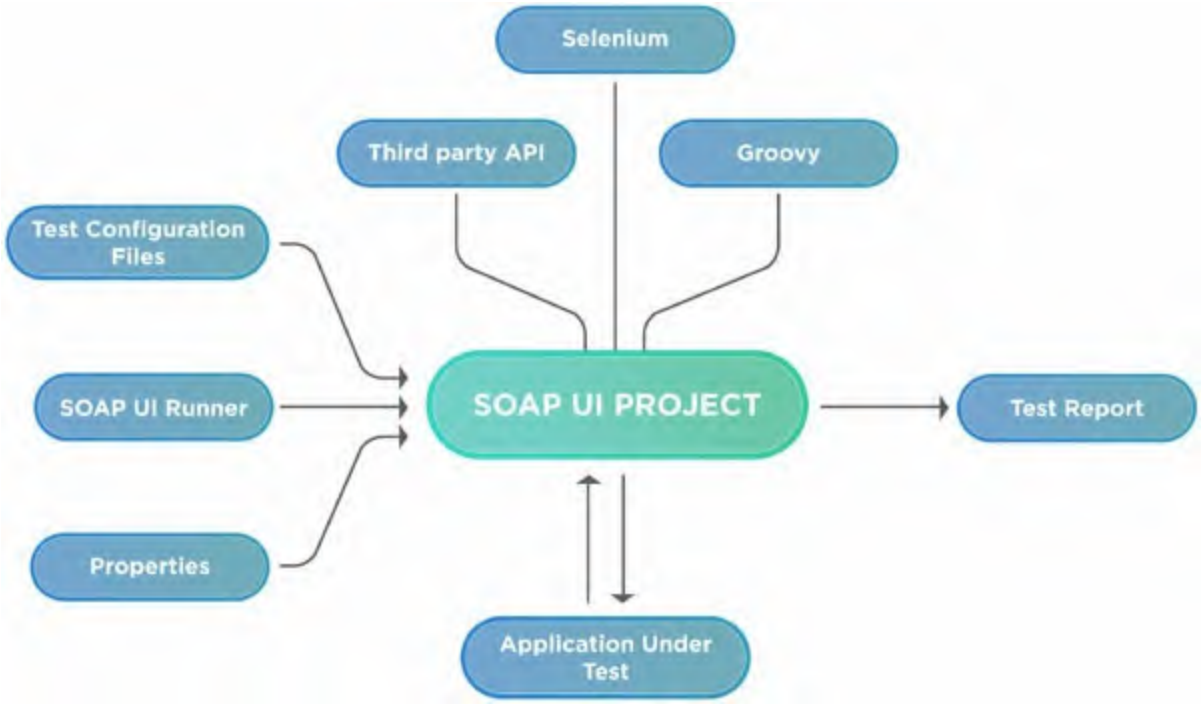

Figure 7.10: *SOAP UI Architecture*

Performance Testing

Performance testing is a method for testing software's responsiveness, scalability, reliability, stability, and resource consumption under high loads. Performance testing is mainly used to identify and fix software application performance issues. It falls under performance engineering and is referred to as "Perf Testing."

This testing generally gets performed whenever a significant change happens in the application and its codebase. Otherwise, those changes can hamper the application's performance, and a slow application is always bad for business.

So, before every major release, the organization conducts performance testing on the new version of the software, and it goes to production only if they feel confident about it.

But performance testing is not only about the application. It is also about the infrastructure that we have provisioned. It checks the performance, reliability, and scaling nature of the infrastructure. The performance testing architecture is shown in the following figure:

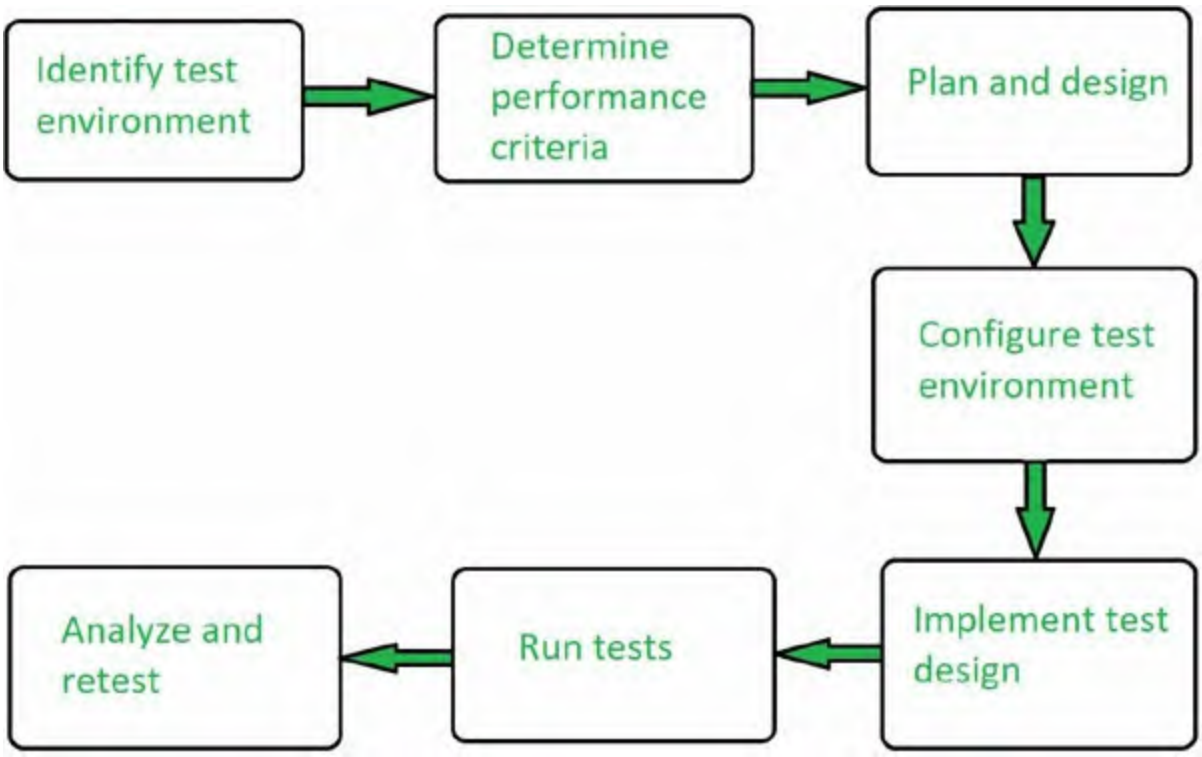

Figure 7.11: *Performance testing architecture*

Abhishek thought of using JMeter for load testing. Multiple tools are available for this, like k6, locust, and so on. He opted for JMeter because it is quite a mature tool with excellent community support. Also, he could record the test results and compare them at every release cycle.

Jmeter

JMeter, commonly referred to as "Apache JMeter," is a graphical, open-source application built entirely on the Java programming language. It is made to evaluate and assess the functional behavior of web applications and a wide range of services during load.

Although JMeter is currently valid in functional testing and in testing JDBC database connections, web services, generic TCP connections, and OS native processes, it is primarily used for testing web applications or FTP applications. To obtain precise performance data for your web server, you can perform various testing activities, such as performance, load, stress, regression, and functional testing. This can be seen in the following figure:

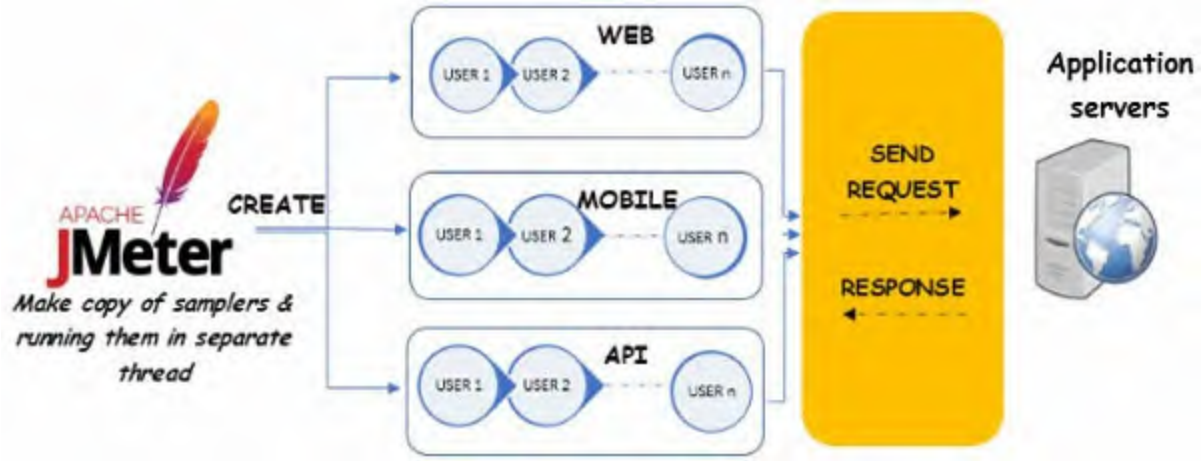

Figure 7.12: *JMETER Architecture*

Figure 7.13 shows a console UI of JMETER:

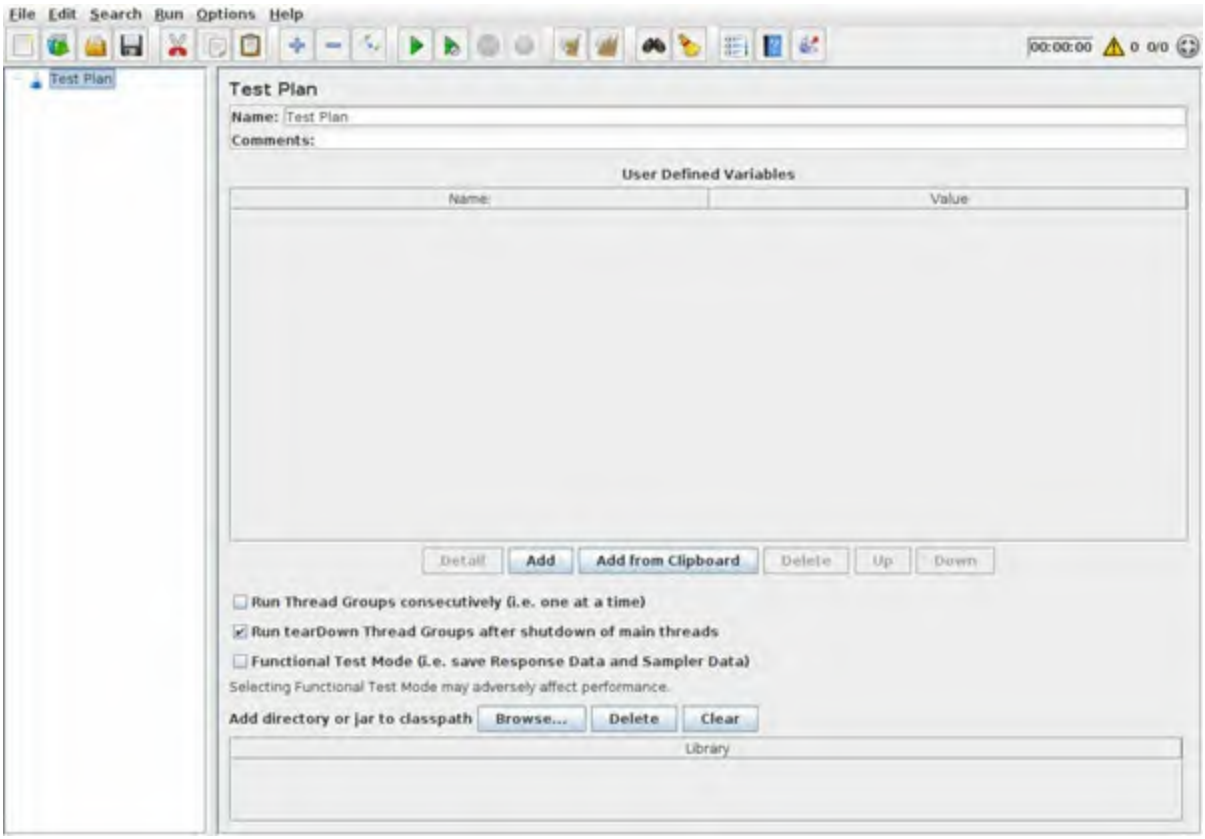

Figure 7.13: *JMETER Console*

Deployment Strategies

Abhishek and the team figured out all the things related to the deployment,

like how many environments are available and how many they need to create, what are the different continuous deployment testing elements, and how to include them.

However, there were still pieces left to complete this deployment jigsaw puzzle, and the major piece was "*Deployment strategies*."

Deployment strategies are different ways of deploying applications according to various use cases. Abhishek knew about the strategies and their names, but he had to evaluate them to perfectly understand and place them into the continuous deployment pipeline.

Some examples of deployment strategies are as follows:

- Normal
- Rolling/Ramped
- Blue Green
- Canary

Normal Deployment

Normal deployment is a dummy deployment that does not do anything specific to transition traffic from version A to version B. In this deployment strategy, we simply remove and deploy version A on the application server.

It is the basic deployment strategy that does not require complex logic for deployment but comes with the cost of downtime. The time fraction between the replacement of version A with version B will cause downtime to the application software, and customers will face issues in this time. This technique implies downtime of the service depending on the shutdown of version A and the bootup time of version B.

Figures 7.13 and *7.14* show that there is a single load balancer serving the traffic. To deploy version 2 of the application, we need to stop the service running on the server and replace the version 1 artifact with that of version 2. After that, we can start the service with the newer code version. Refer to the following figure:

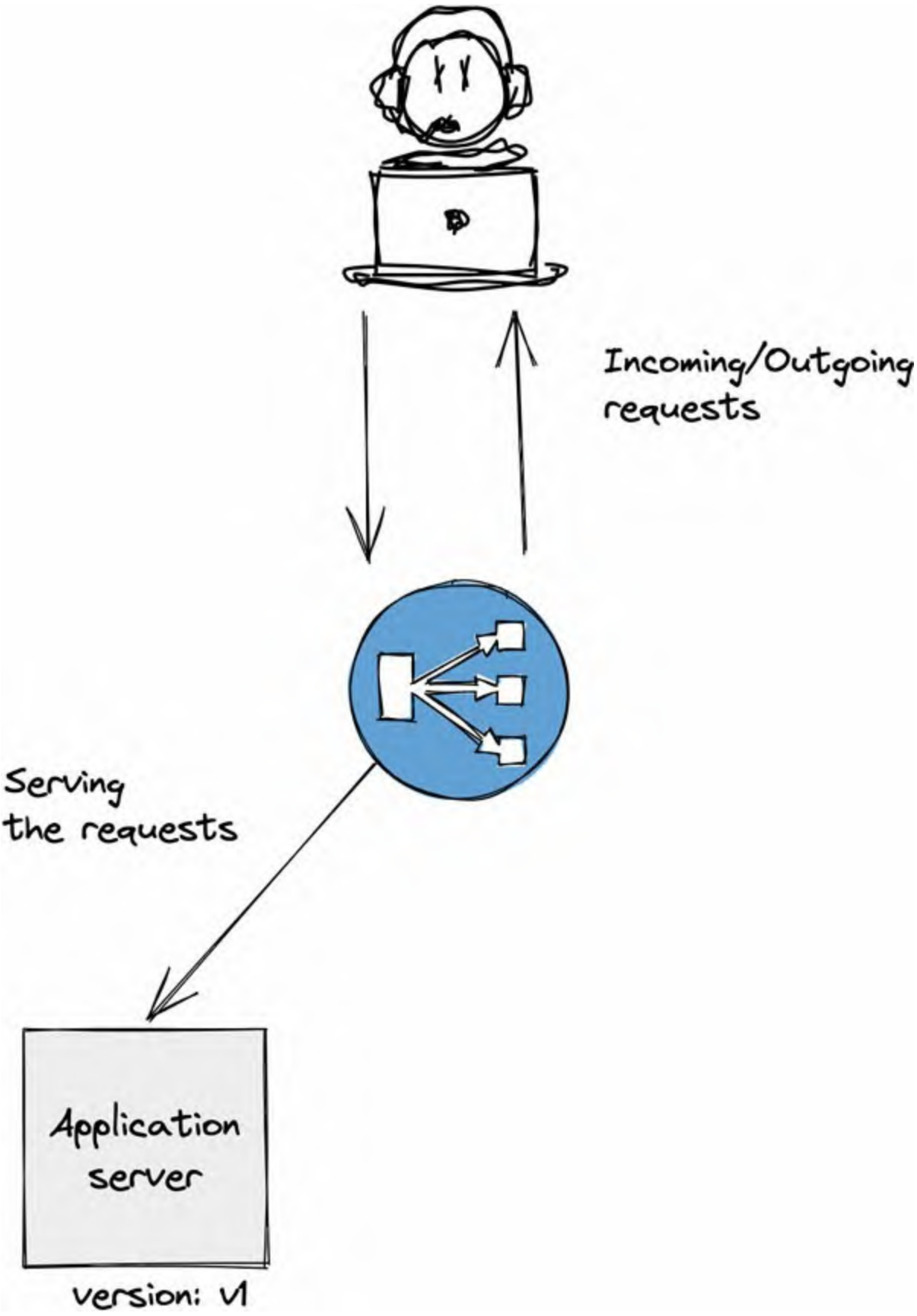
Incoming/Outgoing
requests
Serving
the requests
Application
server
version: v1

Figure 7.14: *Normal Deployment - v1*

Refer to the following figure:

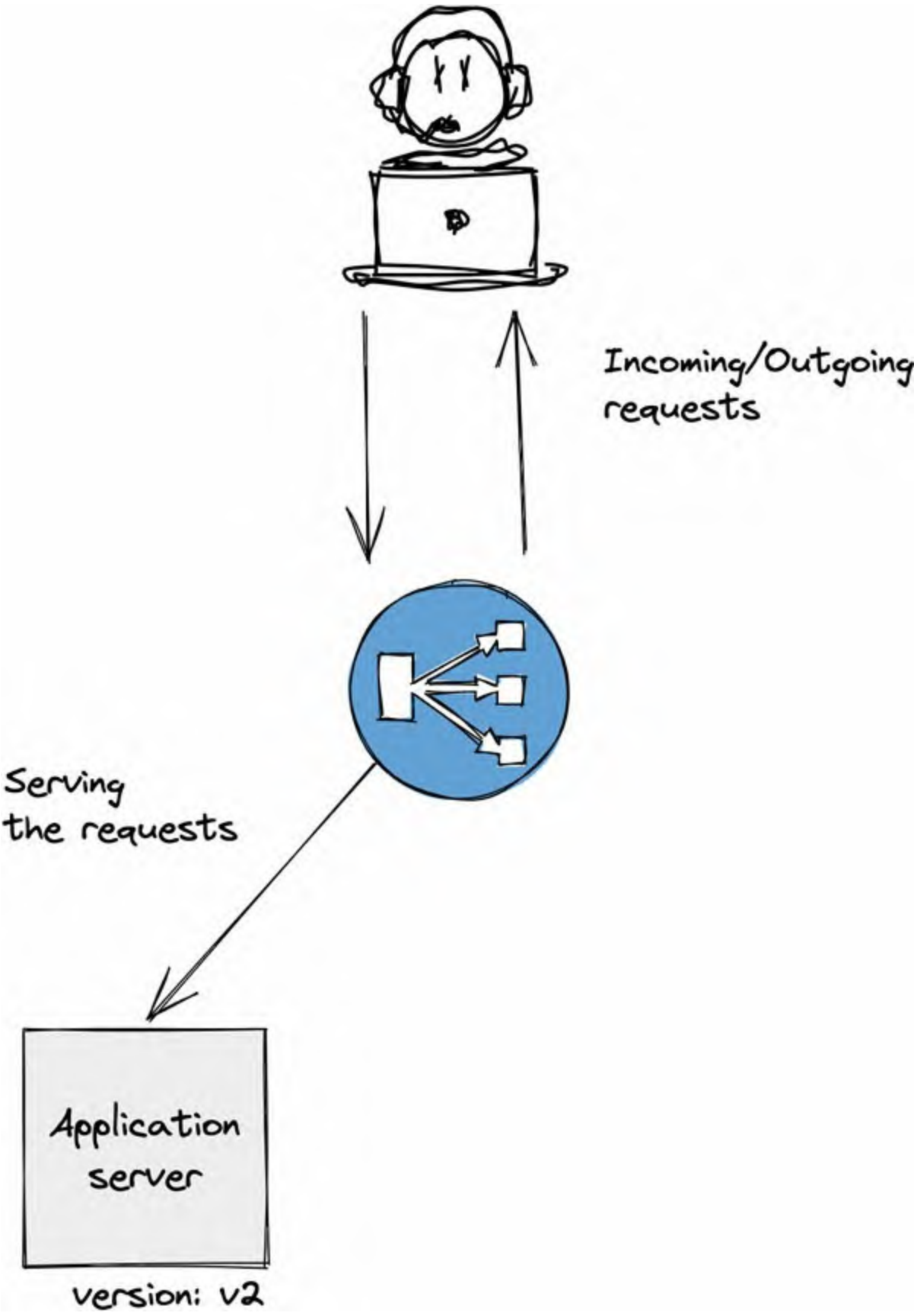
Incoming/Outgoing
requests
Serving
the requests
Application
server
version: v2

Figure 7.15: *Normal Deployment - v2*

Rolling/Ramped Deployment

The ramped/rolling deployment gradually rolls out the version of the application by replacing instances one after another until all instances are rolled out. In this process, the version A instances will be running behind the load balancer, and one instance of version B will also be deployed. When the instance of version B is ready to serve the traffic, it will start receiving the traffic, and one instance from version A will be removed from the load balancer and shut down. This process will continue until all instances of version B are rolled out.

Rolling deployment generally depends on two factors:

- **Max Surge:** This is the number of instances of the new versions that are to be added in the current capacity.
- **Max Unavailable:** This is the number of unavailable instances of the older version in the rolling update procedure, and the number of instances of the older version that need to be removed from the current capacity.

Please refer to the following figures:

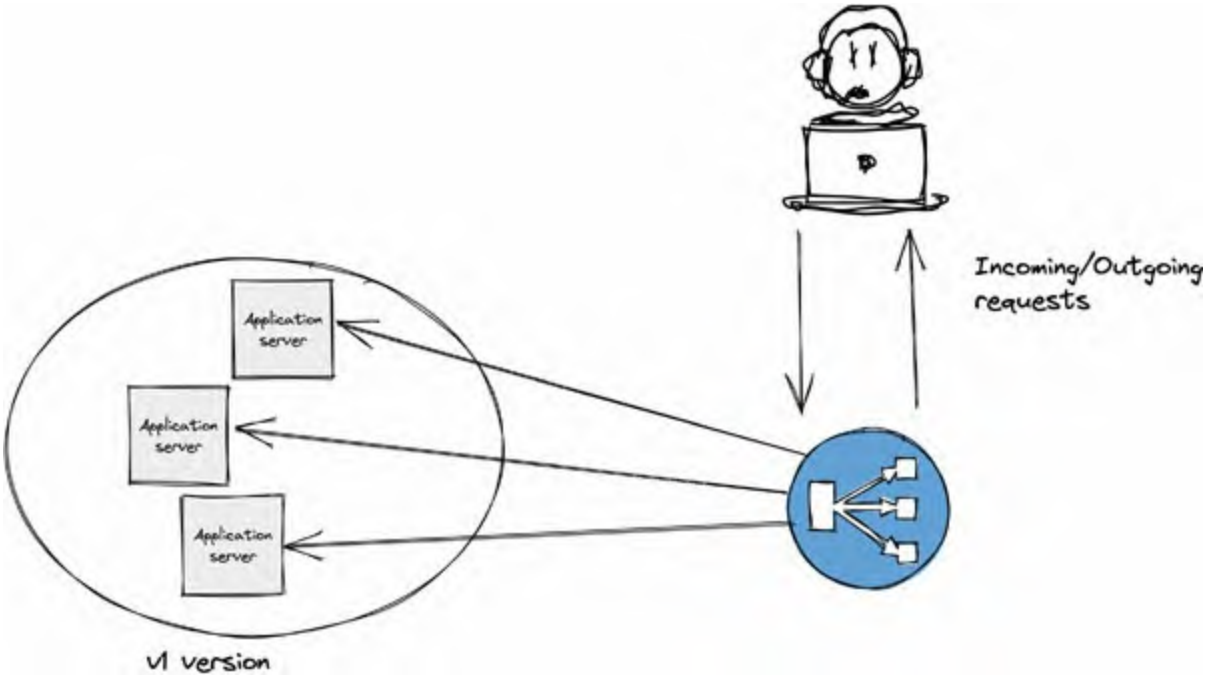

Figure 7.16: *Rolling Deployment - v1*

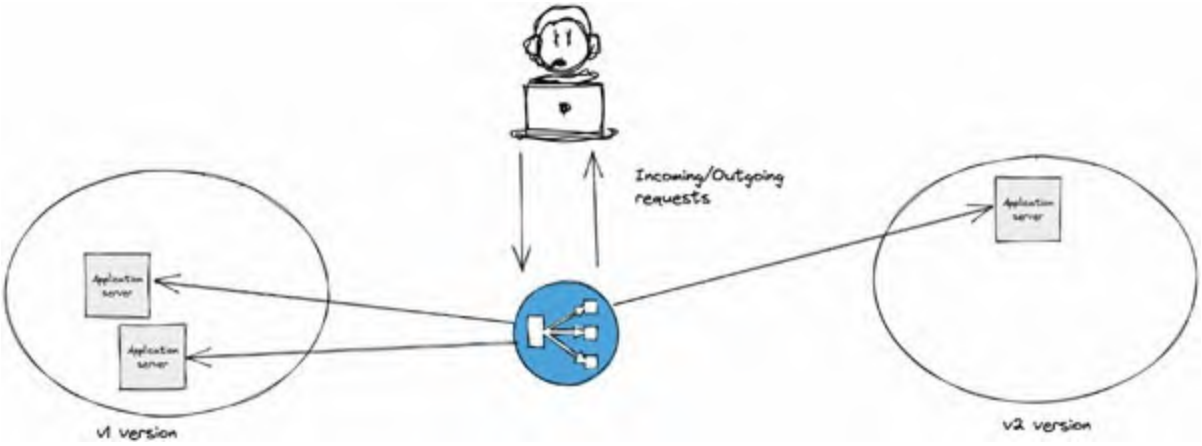

Figure 7.17: *Rolling Deployment - v1 to v2*

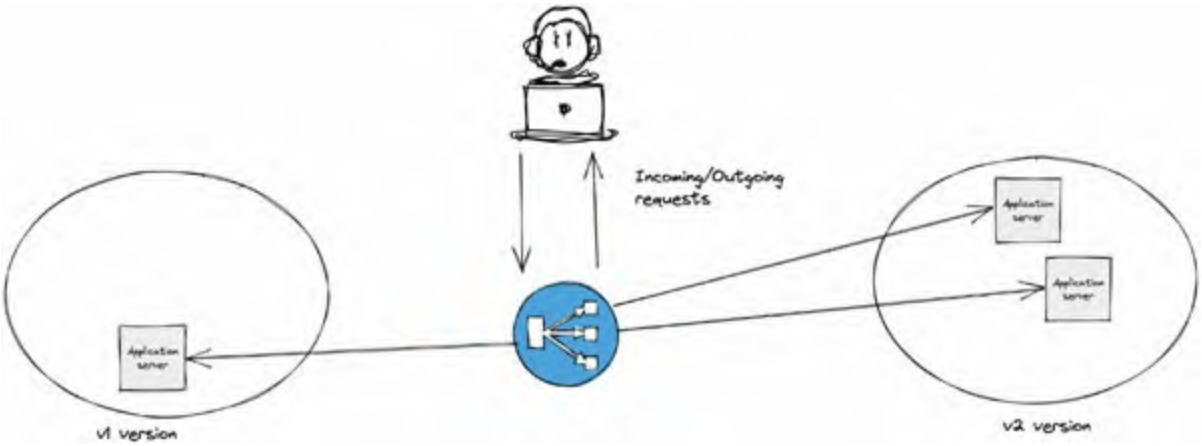

Figure 7.18: *Rolling Deployment - v1 to v2*

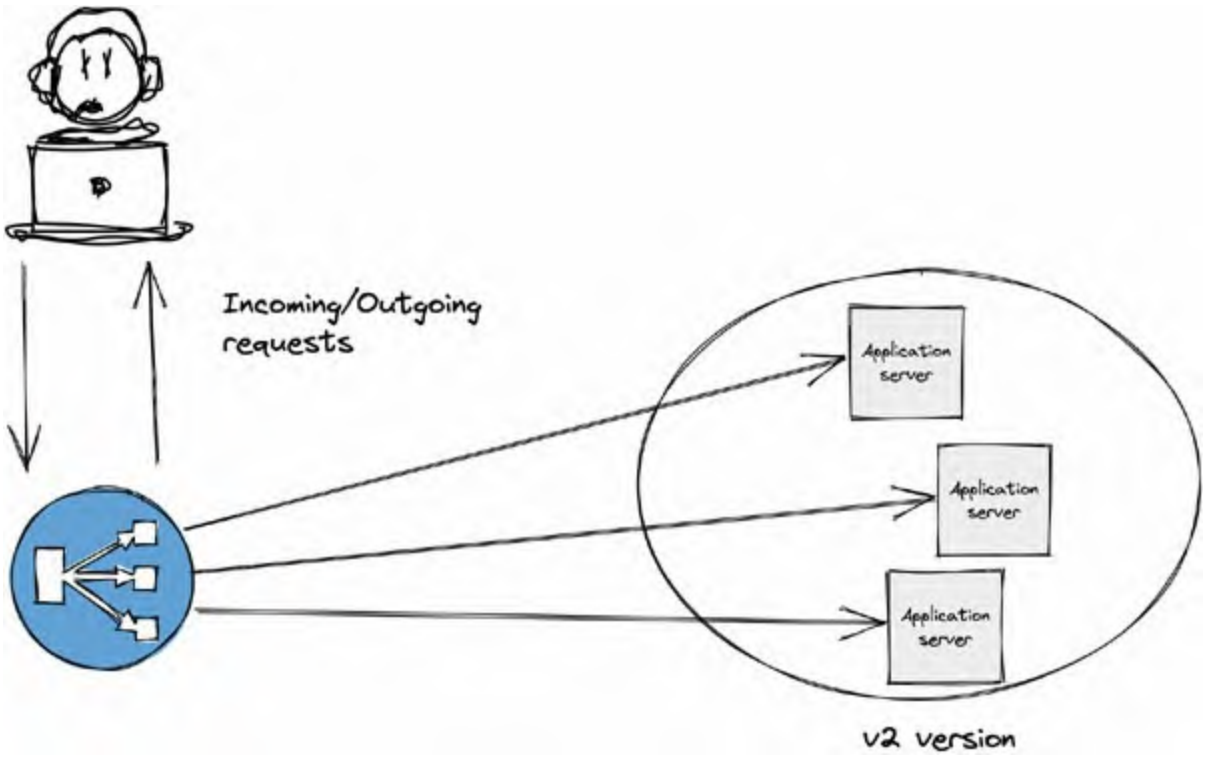

Figure 7.19: *Rolling Deployment - v2*

In the images, we can see how version 2 is being rolled out, and version 1 is getting shut down. One of the major benefits of using this approach is that there will be zero downtime. This is because at any given time, there is at least one server available to serve the requests. The drawback of this strategy is a rollback will be a time-consuming process because it will also use a rolling deployment strategy.

Blue Green deployment

The Blue Green (Red Black) deployment strategy is different from the rolling deployment strategy because in this deployment strategy, the new version of code will be deployed alongside the old version, with the exact number of instances and resources. Once the testing of a newer version of the instance is completed successfully, the traffic is switched from version A to version B on the load-balancer level.

In *figure 7.20* and *7.21*, the flow architecture of Blue Green deployment is shown:

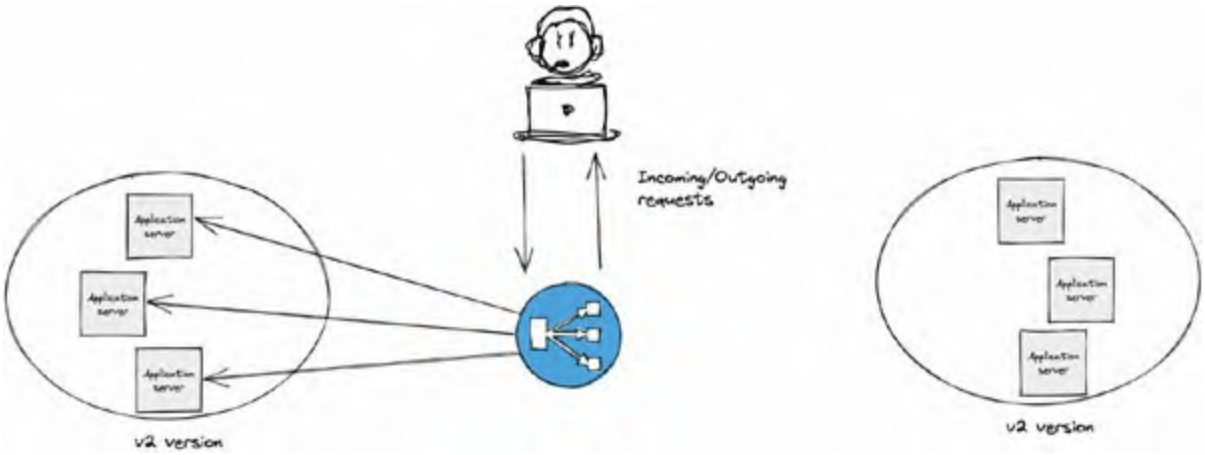

Figure 7.20: *Blue Green Deployment – v1*

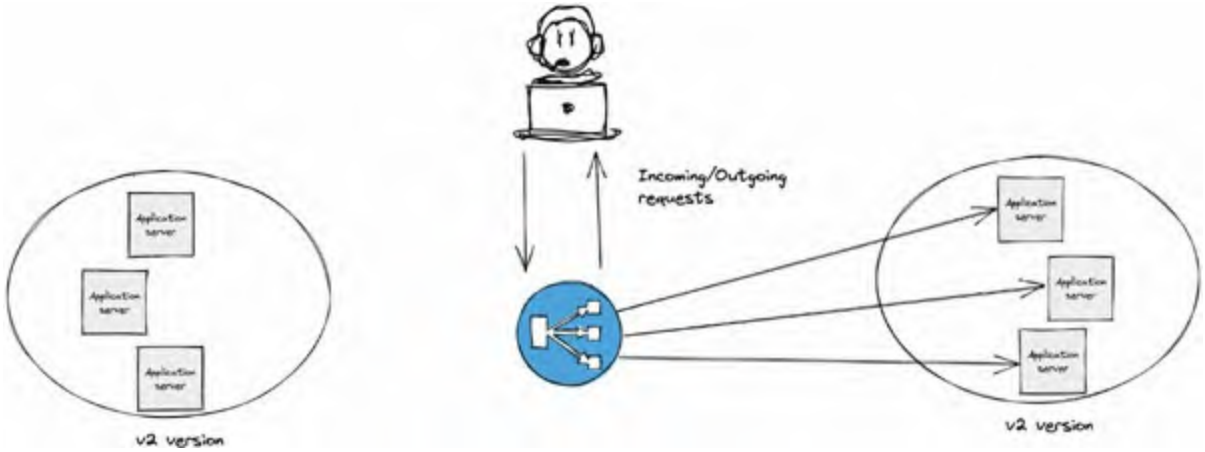

Figure 7.21: *Blue Green Deployment – v2*

The significant advantage of this deployment strategy is that it is quick in traffic switching, so the rollout and rollback of code are much faster than any other deployment strategy, also this strategy does not have downtime. The only drawback is that creating a similar infrastructure with the same resources will cost more because two identical infrastructures will be running in parallel.

Canary Deployment

The Canary Deployment generally shifts the production traffic from version A to version B with live users. In this strategy, we generally use the split based method with weightage. For example, 90 percent of the traffic will be served from version A and 10 percent will be served from version B.

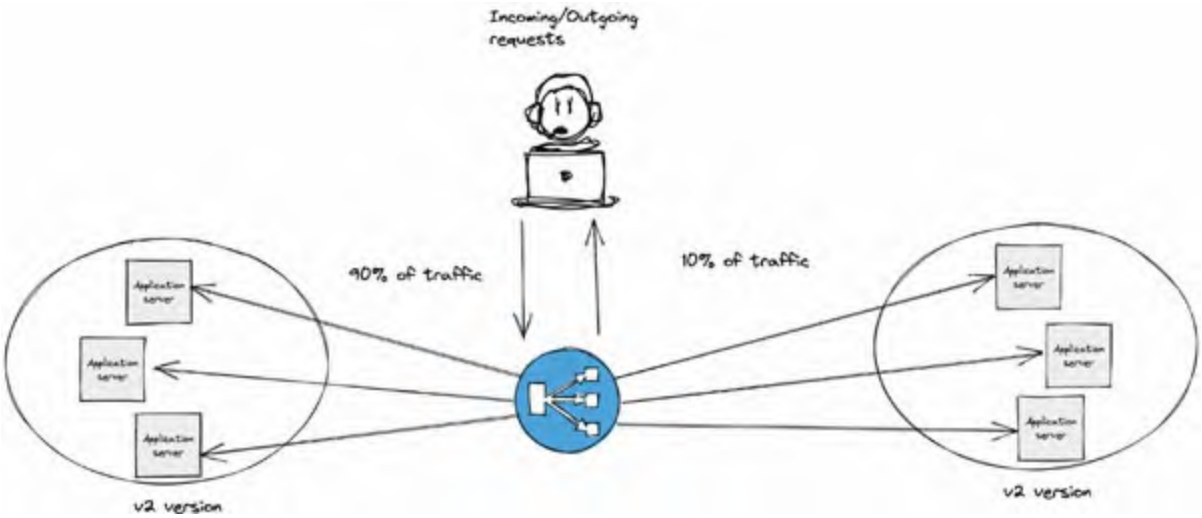

Figure 7.22: *Canary Deployment*

Canary Deployment allows us to capture the performance monitoring insights between versions A and B. This helps us decide whether we want to roll out the newer version. It provides a great capability of a rollback if version B fails, but it is slow in rolling out the newer version. This is illustrated in the following figure:

Strategy	ZERO DOWNTIME	REAL TRAFFIC TESTING	TARGETED USERS	CLOUD COST	ROLLBACK DURATION	NEGATIVE IMPACT ON USER	COMPLEXITY OF SETUP
RECREATE version A is terminated then version B is rolled out	✗	✗	✗	■□□	■■■	■■■	□□□
RAMPED version B is slowly rolled out and replacing version A	✓	✗	✗	■□□	■■■	■□□	■□□
BLUE/GREEN version B is released alongside version A, then the traffic is switched to version B	✓	✗	✗	■■■	□□□	■■□	■■□
CANARY version B is released to a subset of users, then proceed to a full rollout	✓	✓	✗	■□□	■□□	■□□	■■□

Figure 7.23: *Deployment Strategy comparison*

Conclusion

It is time for the final piece of the puzzle. With this study, Abhishek had everything he needed for an end-to-end automated CD pipeline. He had an

understanding of different environments, tests, and deployment strategies. After a successful run of CI pipeline and initial dev/review deployments, the first promotion of CI artifact is to the QA environment. From there on, it is promoted to different environments that were created to suit the specific needs of the project before finally landing in production, where it can impact revenue. Therefore, while creating CD pipelines, things like secrets, environment-specific configuration, artifact promotion, and deployment strategies are considered.

In the next chapter, we will plan and implement environment-specific CD pipelines while going through the technical details. It will provide a clearer picture of CD in action and will be easy to follow for creating your own pipeline.

CHAPTER 8

Continuous Deployment Using Jenkins

Now that the planning of **Continuous Deployment** (**CD**) and research has been completed successfully, the electrifying part of implementation begins. Since we have multiple environments with lots of moving parts, some doubts and concerns must be discussed with all the stakeholders. The next sprint meeting would be the perfect time to lay all those thoughts on the table. In the morning, I made sure to go to the standard break room and let everyone casually know about the progress we had made. After we finish, the entire CI/CD will become a piece of cake. In the past, without automation, they have been through some painful experiences. I enjoyed their lit-up faces, and they piqued my interest.

Structure

In this chapter, we will discuss the following topics:

- Deployment strategy discussion
- Continuous Deployment for QA Environment (Normal Deployment)
- Continuous Deployment for Security Environment (Rolling Deployment)
- Continuous Deployment for Performance Environment (Blue/Green Deployment)
- Continuous Deployment for UAT Environment (Canary Deployment)
- Continuous Deployment for Production Environment (Canary Deployment)
- Reflection

Objectives

After going through this chapter, you should be able to use different kinds of

deployments in your CD and decide on the right kind of deployment based on the use case. This chapter will also help you understand the different organizational environments' importance. As an outcome, you should be able to implement end-to-end CD pipeline using Jenkins with automation.

Deployment strategy discussion

Once the meeting started, everyone grabbed their coffee and was seated in their respective seats. I began to explain the things I had learned while studying continuous deployment.

Sajal: "Abhishek, what do you think, how many environments will be needed in our scenario?"

Me: "Well, considering all things, there should be at least five environments."

Sajal: "Whoa! Five environments! That's quite a significant number, don't you think?"

Me: "Yeah, but every environment has its importance, and I don't think we can neglect any of them."

Adeel: "Hmm, interesting**.** What are the environments that we are including?"

Me: "We will include QA, security, performance testing, UAT, and production."

Adeel: "I know the QA and production environments are important, but since you're advocating for others, I am interested to know what they're about."

Me: "Okay, so here's what I know**.** We need to have these environments because in the QA environment, we will only be validating the functionality of our services, but we must have a dedicated environment to target security precisely. Here, we will do rigorous testing, like **Dynamic Application Security Testing (DAST)**, exposing all the vulnerabilities before going live. This will also give us time to fix them."

Sajal: "I agree that security can't be neglected, and we must treat it as a priority."

Me: "Yes, the performance testing environment is needed to check the application performance after any significant release. While adding features, we cannot compromise on the application's integrity, so we need an

environment where we can test the application in terms of responsiveness."

Adeel: "I didn't think about it. Good point. What about the UAT environment?"

Me: "Yeah, that is also important because most of our tests will be from an application developer's perspective on a minimal infrastructure. We need to replicate and set up an environment close to production where the application can also be tested from the users' perspectives. So, when we release the code, people will not face issues, and our reputation will not be trampled."

Sajal: "That's right. It looks like you have given a lot of thought to it. By the way, have you given any thought to the deployment strategies? What type of deployment strategy we are going to use?"

Me: "To be honest, it will be a mix of several deployment strategies in different environments. Since each deployment strategy has its purpose, we must select the environment and deployment strategy, respectively."

Adeel: "Knowing you, I think you have already thought it through. Lay it on us."

Me: "Well, you know me very well, I guess! I have given some thought to it. In my opinion, we can go with standard deployment in a QA environment."

Sajal: "Why normal deployment in QA? Is there any rationale behind it?"

Me: "Yes, there is a rationale; so in normal deployment, there will be downtime, and I don't think that will impact the QA environment in any way since it's in an internal environment, and no one is going to access it from outside. I think normal deployment should be fine for the this environment."

Adeel: "Well, it makes sense to me. A little downtime is not going to harm the QA environment."

Me: "Correct. For the security testing environment, I think rolling deployment should be good. The need for keeping rolling deployment here is that most security testing will be automated, and it will run for an extended period. Therefore, we cannot risk downtime in the security testing environment; otherwise, the tests will be disrupted."

Sajal: "Yeah, that sounds perfect. We don't want our security testing to get hampered."

Me: "The only drawback is that the rollback will be a little slow in rolling deployment, but I think we can manage that. In the performance testing

environment, we should use the Blue/Green deployment strategy to cut over the traffic easily and quickly on the newly released version and then perform rigorous testing for performance. If anything looks bad, we will cut back to the older versions. The rollback is quite fast in this deployment strategy."

Adeel: "Hmm. It sounds nice, but I think we need to pay a few extra dollars for this deployment strategy."

Sajal: "Yes, Adeel, but we cannot compromise our client user experience and performance. So, let us go ahead with it."

Adeel: "Abhishek, what about UAT and production? Since they are close, will they follow the same deployment strategy?"

Me: "That is a great question, and the answer is "Yes". We will follow the canary deployment approach for UAT and the production environments."

Sajal: "Abhishek, why canary? Do we want to test the actual user experience there?"

Me: "Yes, sir, you are right. We will shift some percentage of traffic on the new release and then compare the graph, metrics, and user experience. If everything looks good, we will gradually shift the traffic to the new version, and the old version will have 0% traffic eventually.

Sajal: Looks like we got everything covered. I like the canary idea. Let's try it out. So, Abhishek, please start working on the solutions you have explained, and we will see how everything works. Please let me know if you need anything from my side."

Me: "I will surely let you know."

Adeel: "I can also help if you need anything from my side or any specific development practice changes from the development team."

Me: "I appreciate that, Adeel. Although I doubt it will be required, I will surely let you know in case."

The following figure shows the documentation of all the flows for better understanding:

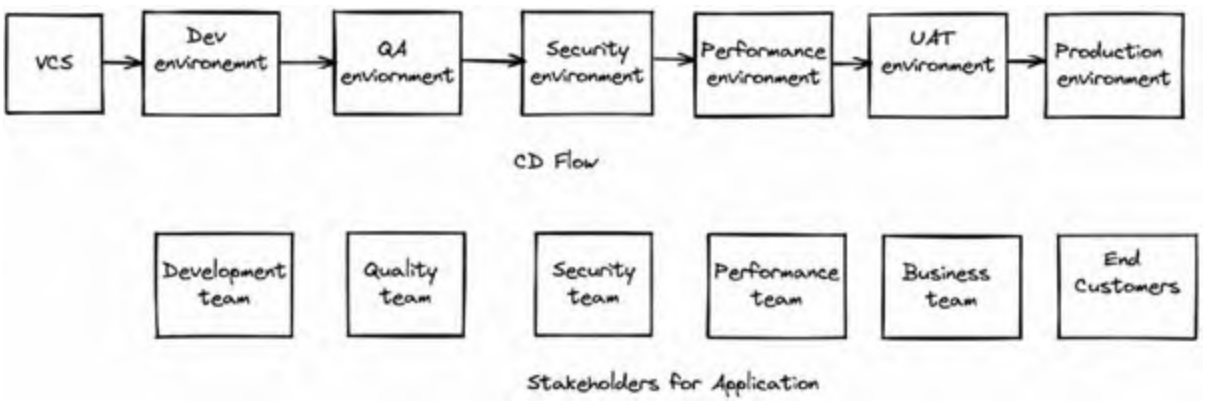

Figure 8.1: *Flow for the complete CD with environments*

The preceding figure shows the complete flow of CD across different environments. The first deployment will happen on the Development environment, and the development team will be the owner in terms of validation. After the development team approves the code and merges it with the master branch, the code will be deployed to the QA environment, and the quality team will validate it.

After QA signoff, the code will move to the security environment, where the security team will perform all the security-related testing. If there are no vulnerabilities, the code will move to the performance testing environment, where the performance testing team will test the application's performance and responsiveness.

At the end of the phase, the code will be deployed in the UAT environment and validated by the business team for user perspective testing. If everything looks fine, the code will be deployed in the production environment.

Continuous Deployment for QA Environment (Normal Deployment)

Figure 8.2 shows the flow diagram of the QA environment:

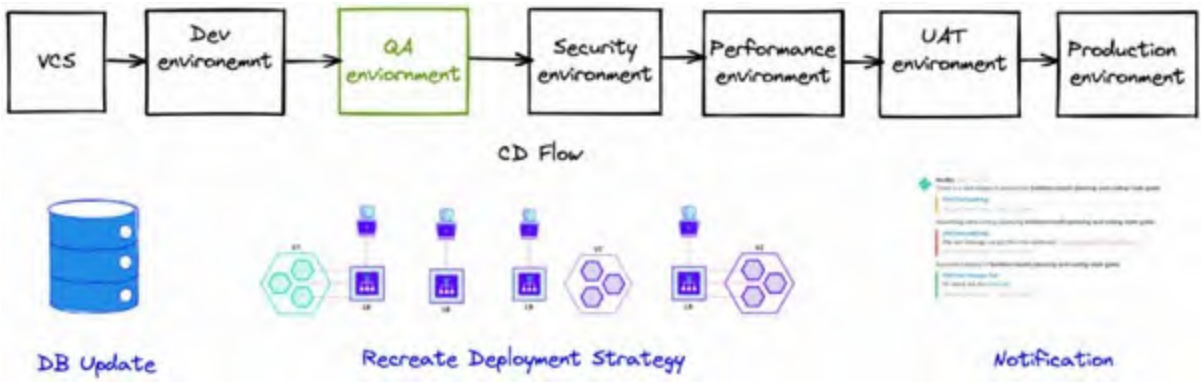

Figure 8.2: *Flow for QA Environment*

Figure 8.3 is the graphical representation of QA environment deployment:

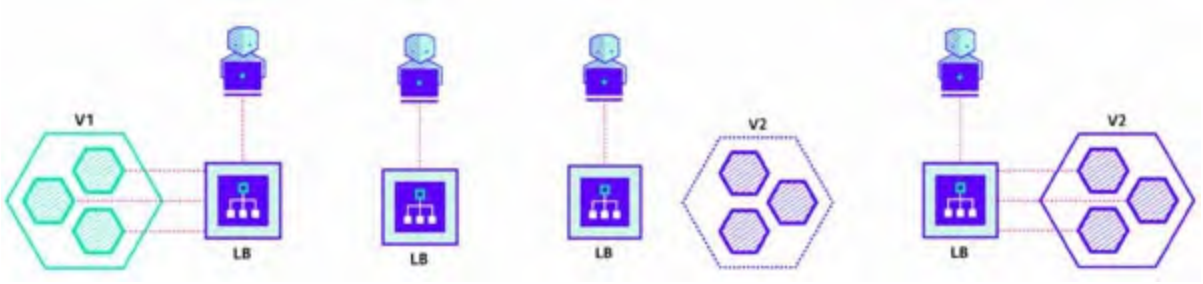

Figure 8.3: *Recreate strategy for QA Environment*

As discussed in sprint planning, the QA environment will use Recreate/Normal rolling deployment strategy, in which the old code will be removed. New code will be deployed under the same load-balanacer.

To set up the environment, we need to create a load-balancer that will serve the traffic on the application. For load-balancer, we will use **Traefik**, which is a modern advanced load balancer and can be integrated with Docker very easily. To create this load balancer, we can execute the docker command or the docker-compose command:

```
version: "3.3"
services:
  traefik:
    image: "traefik:v2.8"
    container_name: "traefik"
    command:
      #- "--log.level=DEBUG"
      - "--api.insecure=true"
      - "--providers.docker=true"
      - "--providers.docker.exposedbydefault=false"
      - "--entrypoints.web.address=:80"
    ports:
```

```
    - “80:80”
    - “8080:8080”
  volumes:
    - “/var/run/docker.sock:/var/run/docker.sock:ro”
  network_mode: bridge
```

Code: 8.1

After that, we can simply execute the docker-compose command to bring it up:

```
$ docker compose up -d
```

Code 8.2

The stage view pipeline for deployment in the QA environment will look like this:

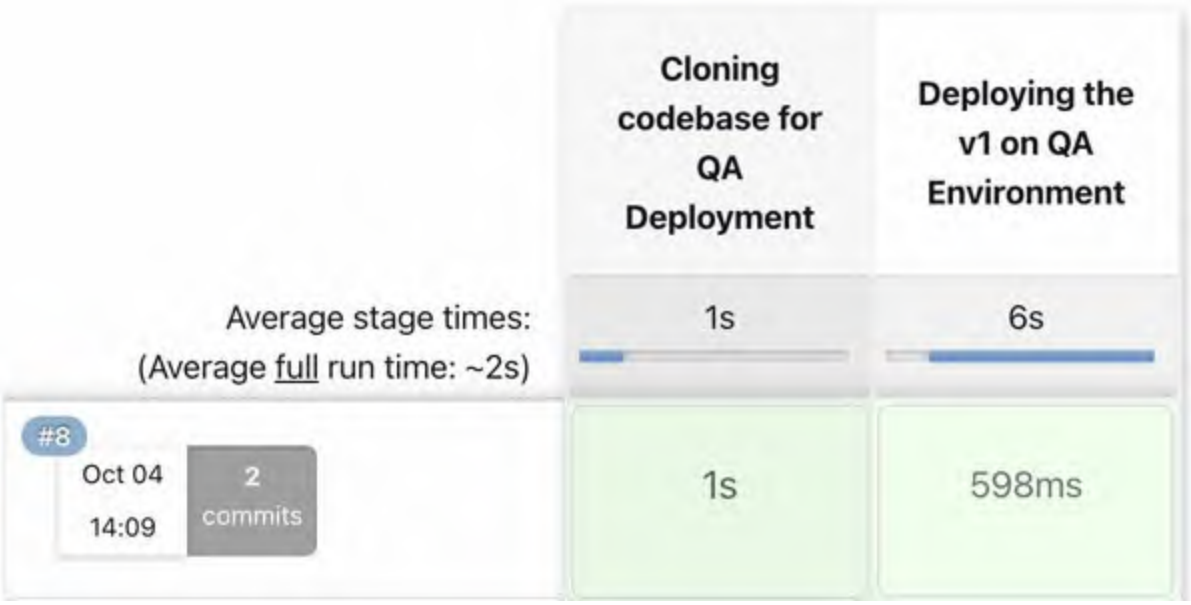

Figure 8.4: *Pipeline view for QA environment*

The Jenkinsfile will be the orchestrator for creating this stage view. The Jenkinsfile code will look like this:

```
node(“deployment”) {
  properties([parameters([string(defaultValue: ‘latest’, name:
  ‘VERSION’, trim: true)])])
  stage(“Cloning codebase for QA Deployment”) {
     checkout scm
  }
  stage(“Deploying the ${VERSION} on QA Environment”) {
     sh “””
     if ! sudo docker inspect spring3hibernate > /dev/null 2>&1
```

```
    then
     sudo docker run -itd --name spring3hibernate --label
     traefik.
      enable=true \
     --label
     'traefik.http.routers.spring3hibernate.rule=Host(`qa-
      spring.opstree.com`)' \
     --label "traefik.port=8080" opstree/
      spring3hibernate:${VERSION}
    else
     sudo docker rm -f spring3hibernate
     sudo docker run -itd --name spring3hibernate --label
     traefik.
      enable=true \
     --label
     'traefik.http.routers.spring3hibernate.rule=Host(`qa-
      spring.opstree.com`)' \
     --label "traefik.port=8080" opstree/
      spring3hibernate:${VERSION}
    fi
    """
  }
}
```

Code 8.3

In the code example, we are using the concept of labels for docker containers to use Traefik service discovery model. More information is available at the following link:

[https://doc.traefik.io/traefik/routing/providers/docker/]

The architecture diagram of Traefik with a docker container looks like this:

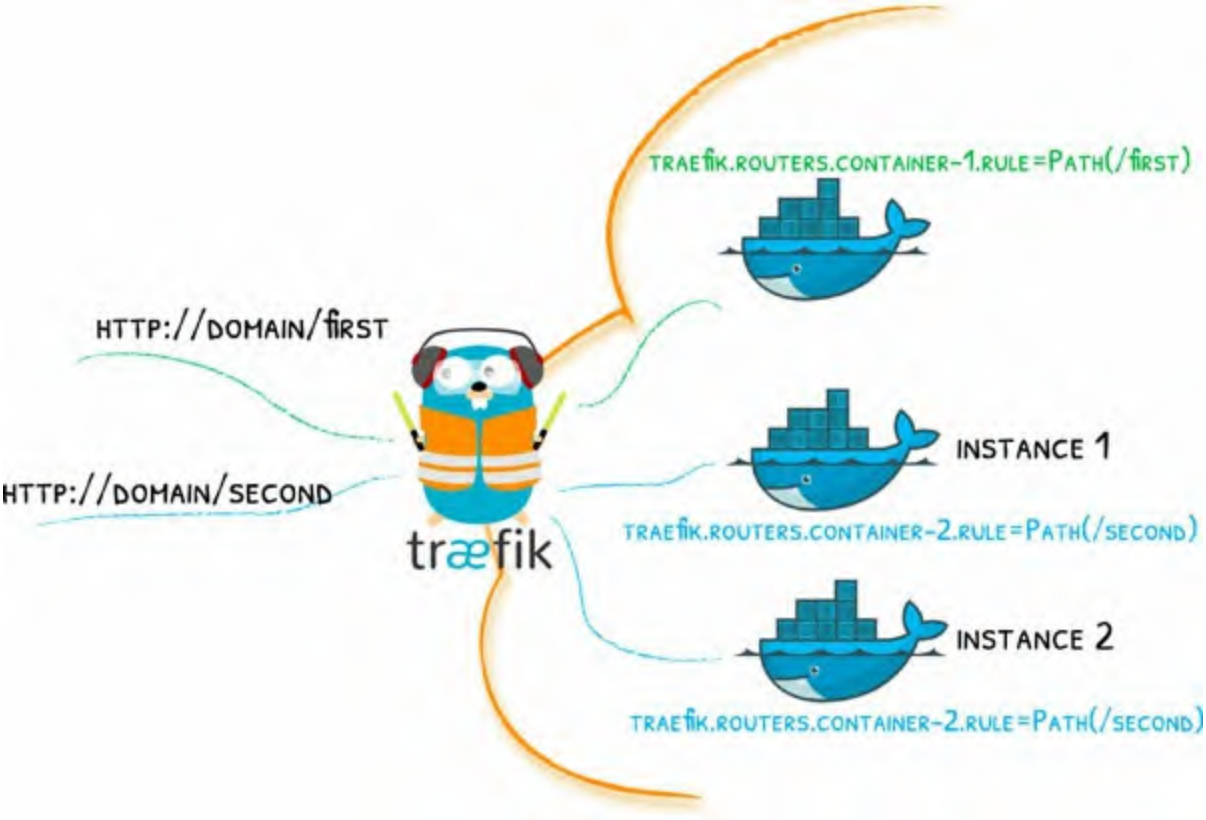

Figure 8.5: *Traefik docker architecture*

Continuous Deployment for Security Environment (Rolling Deployment)

In the following figure, you can see the flow diagram of the security environment:

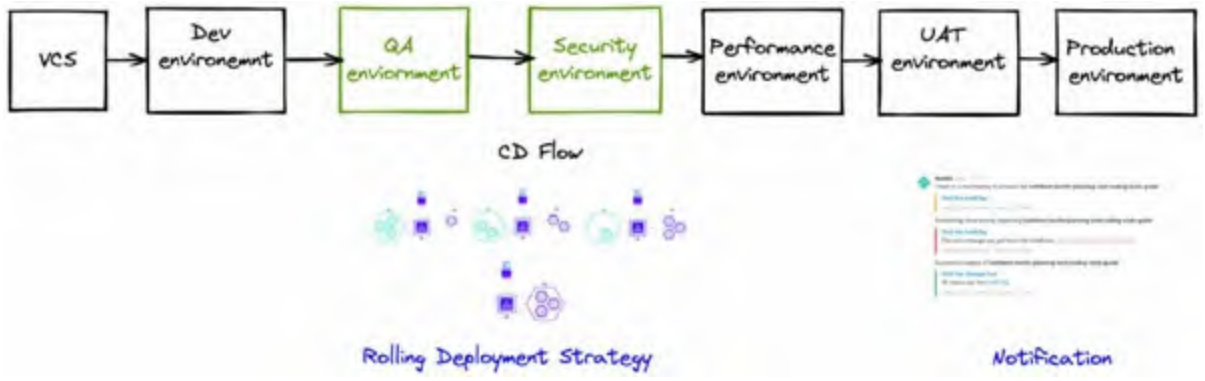

Figure 8.6: *Flow diagram of Security Environment*

Figure 8.7 is the graphical representation of security environment

deployment:

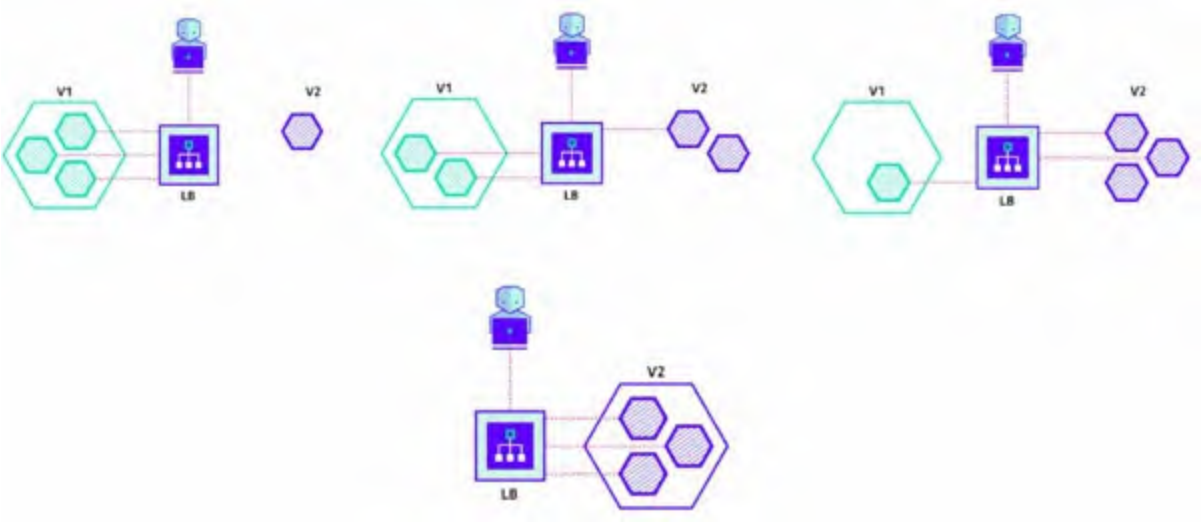

Figure 8.7: *Rolling Deployment for Security Environment*

To make sure we are not disrupting the security testing by giving some application downtime, we are going to use rolling deployment strategy. In this strategy, the v2 version of the code will be deployed alongside the v1 version, and once the health check is passed for the v2 version, we will remove the v1 version from the load balancer and delete it.

The stage view of the security environment with rolling deployment will look like this:

Figure 8.8: *Stage view for Security Environment*

The Jenkinsfile code for a rolling deployment will look like this:

```
node(“deployment”) {
  properties([parameters([string(defaultValue: ‘latest’, name:
  ‘OLDER_VERSION’, trim: true),
```

```
  string(defaultValue: 'latest', name: 'NEW_VERSION', trim:
  true)])])
  stage("Cloning codebase for Security Environment Deployment") {
     checkout scm
  }
  stage("Deploying the ${NEW_VERSION} on Security Environment") {
     sh """
     sudo docker run -itd --name spring3hibernate-${NEW_VERSION}
     opstree/spring3hibernate:${NEW_VERSION}
     """
  }
  stage("Validating the ${NEW_VERSION}") {
     sh """
     sleep 10s
     NEW_IP=\$(sudo docker inspect -f
     '{{range.NetworkSettings.Networks}}{{.IPAddress}}{{end}}'
     spring3hibernate-${NEW_VERSION})
     response=\$(curl --write-out '%{http_code}' --silent --
     output /dev/null http://\${NEW_IP}:8080)
     if [[ "\${response}" != 200 ]; then
      echo "Application is not working fine"
      exit 1
     fi
     """
  }
  stage("Add the new version to load balancer") {
     sh """
      sudo docker rm -f spring3hibernate-${NEW_VERSION}
      sudo docker run -itd --name spring3hibernate-${NEW_VERSION}
      --label traefik.enable=true \
      --label
      'traefik.http.routers.spring3hibernate.rule=Host(`qa-
      spring.opstree.com`)' \
      --label "traefik.port=8080"
      opstree/spring3hibernate:${NEW_VERSION}
    """
  }
  stage("Removing old version") {
     sh """
     sudo docker rm -f spring3hibernate-${OLDER_VERSION}
     """
  }
}
```

Code 8.4

In this pipeline, we are deploying a new Docker container with a new version of application image; once the container is deployed, there is another stage to

validate if the new version is healthy and working fine. Once the validation is complete, we will add the new version to the load balancer, and after that, the older version will be removed from the load balancer and the server.

Continuous Deployment for Performance Environment (Blue/Green Deployment)

In the following figure, you will see the flow diagram of the performance environment:

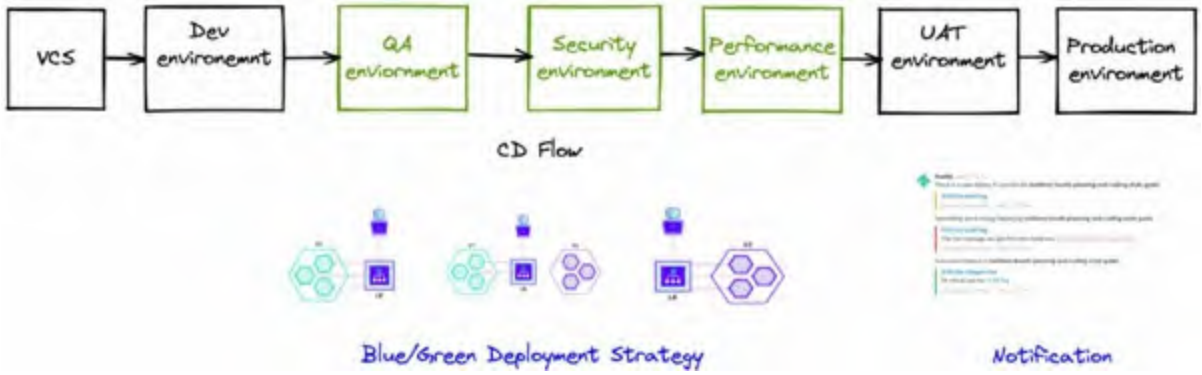

Figure 8.9: *Flow diagram of the Performance Environment*

Figure 8.10 is a graphical representation of the performance environment's deployment:

Figure 8.10: *Blue Green Deployment of the Performance Environment*

In the performance testing environment, we will do the blue-green deployment so that cutover will be quick to the new environment. If anything doesn't work, rollback will also be quick.

Verion 2 of the code will be deployed along with version 1, and after testing and validation, the traffic will be cut over to version 2. After traffic validation, version 1 can be stopped, and if anything doesn't go well with version 2, we can perform the rollback action.

The stage view of the pipeline for rollback look like this:

Figure 8.11: *Blue Green Deployment Stage View - Rollback*

The stage view of the pipeline for making the traffic live will look like this:

Figure 8.12: *Blue Green Deployment Stage View - Proceed*

The pipeline code for Blue Green deployment will look like this:

```
node("deployment") {
  properties([parameters([
     string(defaultValue: 'latest', name: 'GREEN_VERSION', trim:
     true, description: 'Green is newer version'),
     string(defaultValue: 'latest', name: 'BLUE_VERSION', trim:
     true, description: 'Blue is older version')])])
  stage("Cloning codebase for Security Environment Deployment") {
     checkout scm
  }
  stage("Deploying the Green version on PT Environment") {
     sh """
     sudo docker run -itd --name spring3hibernate-green --label
     traefik.enable=true \
     --label
     'traefik.http.routers.spring3hibernate.rule=Host(`green-pt-
     spring.opstree.com`)' \
```

```
    --label “traefik.port=8080”
    opstree/spring3hibernate:${GREEN_VERSION}
    “””
  }
  stage(“Pause pipeline for Green version validation”) {
     input ‘Do you want to proceed further?’
  }
  stage(“Cut-over of traffic on Green version from Blue version”)
  {
     sh “””
     aws route53 change-resource-record-sets --hosted-zone-id
     Z10166071X5VNARP9PUCV --change-batch \
     file://blue-green-performance/green-update.json
     “””
  }
  stage(“Validation of Performance Testing Environment”) {
     returnValue = input message: ‘Do you want to proceed
     further?’,
     parameters: [choice(choices: [‘Rollback’, ‘Proceed’], name:
     ‘action’)]
}
  if (“${returnValue}” == “Rollback”) {
     stage(“Rollback in case of failure”) {
       sh “””
       aws route53 change-resource-record-sets --hosted-zone-id
       Z10166071X5VNARP9PUCV --change-batch \
       file://blue-green-performance/blue-update.json
       “””
     }
  } else if (“${returnValue}” == “Proceed”) {
     stage(“Live the traffic”) {
       sh “””
       sudo docker rm -f spring3hibernate-green
       sudo docker run -itd --name spring3hibernate-blue --label
       traefik.enable=true \
       --label
       ‘traefik.http.routers.spring3hibernate.rule=Host(`blue-pt-
       spring.opstree.com`)’ \
       --label “traefik.port=8080”
       opstree/spring3hibernate:${GREEN_VERSION}
    “””
     }
  }
}
```

Code 8.5

Continuous Deployment for UAT Environment (Canary Deployment)

Figure 8.13 shows the flow diagram of the UAT environment:

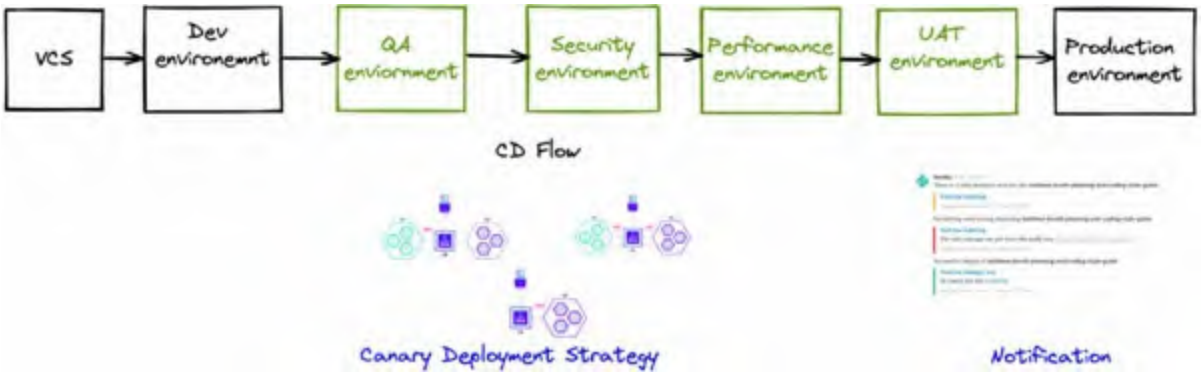

Figure 8.13: *Flow diagram of the UAT Environment*

Figure 8.14 shows the graphical representation of UAT environment deployment:

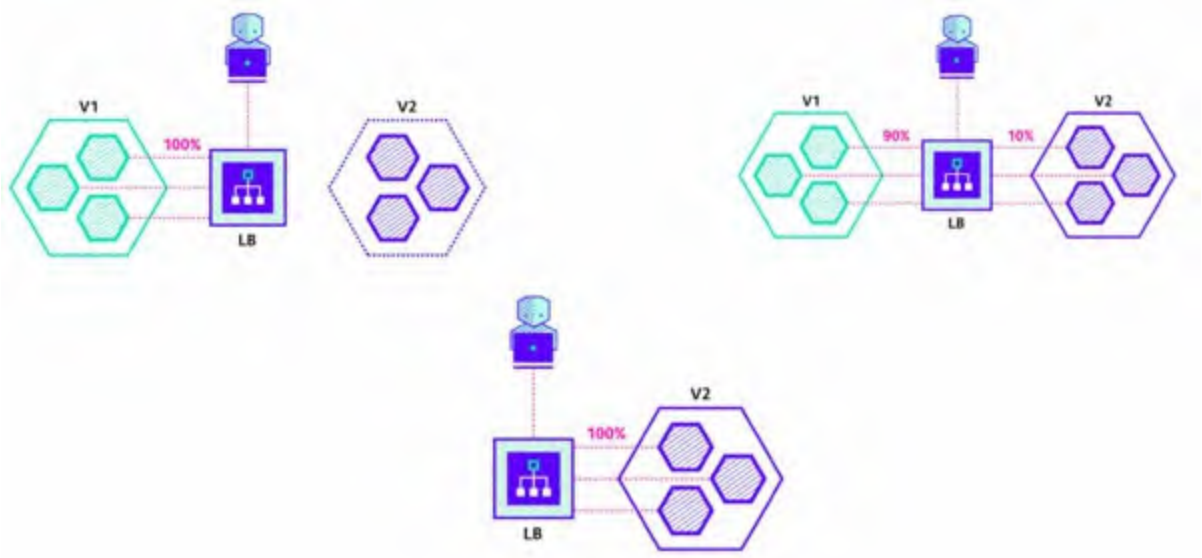

Figure 8.14: *Canary Deployment for UAT Environment*

In the UAT environment, the application will be deployed using the canary deployment strategy as described in sprint planning. Version 2 will be deployed alongside version 1, and the traffic will be shifted in percentage to version 2. 90% of traffic will be served from version 1, and the remaining traffic will be served from version 2. If all goes well, the traffic will completely go to version 2, and version 1 will be torn down. In case of

failure, 10% of the traffic will again be shifted to version 1.

The stage view of the UAT canary deployment pipeline for making traffic live will look like this:

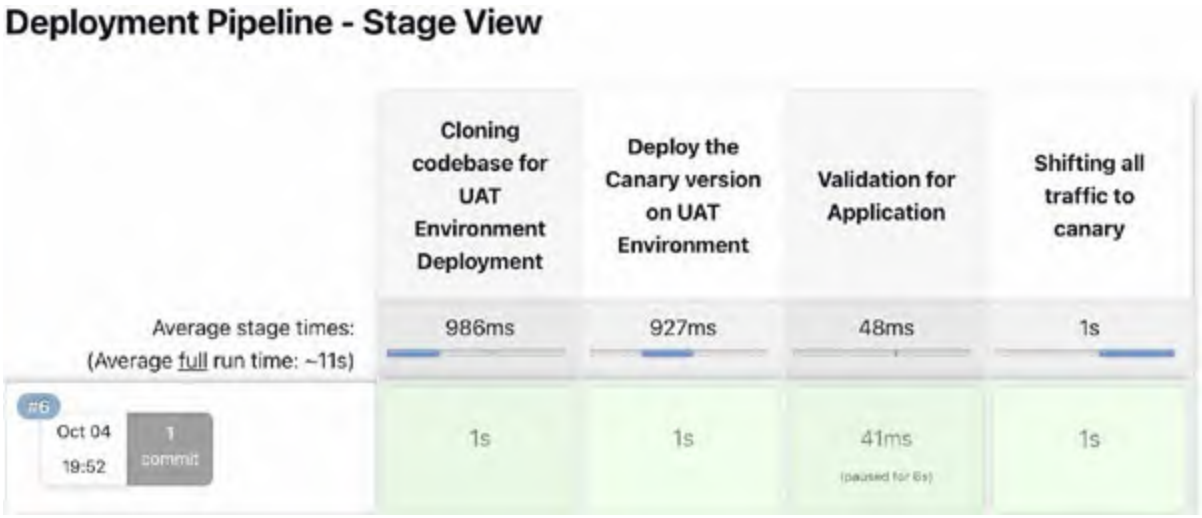

Figure 8.15: *Canary Deployment Stage View – Live*

The stage view of the UAT canary deployment pipeline for rollback will look like this:

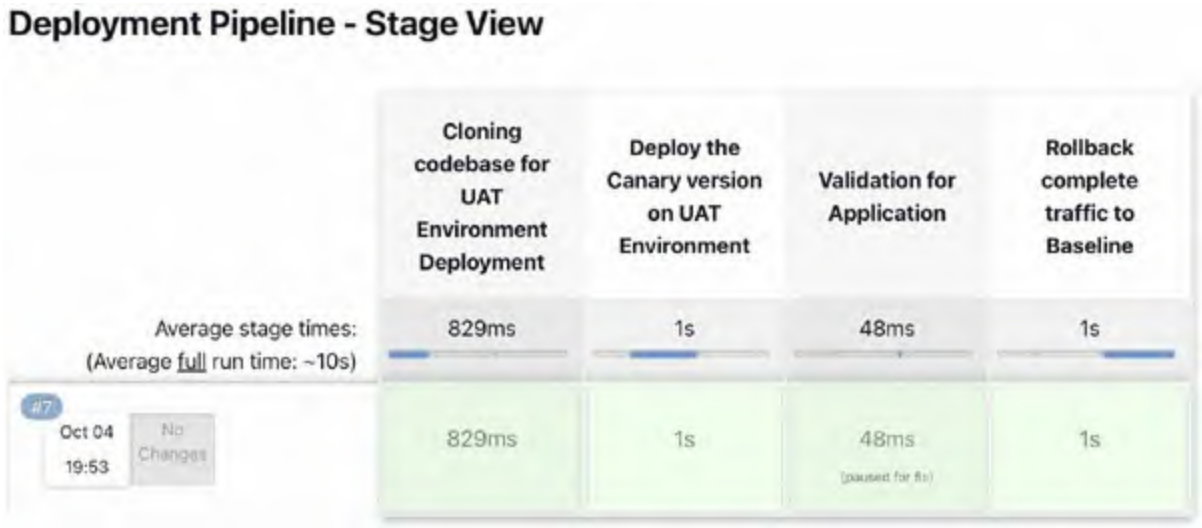

Figure 8.16: *Canary Deployment Stage View - Rollback*

The pipeline code for canary deployment will look like this:

```
node("deployment") {
  properties([parameters([
     string(defaultValue: 'latest', name: 'BASELINE_VERSION',
     trim: true, description: 'Older version'),
     string(defaultValue: 'latest', name: 'CANARY_VERSION', trim:
```

```
   true, description: 'Newer version')])])
stage("Cloning codebase for UAT Environment Deployment") {
   checkout scm
}
stage("Deploy the Canary version on UAT Environment") {
   sh """
   sudo docker run -itd --name spring3hibernate-canary --label
   traefik.enable=true \
   --label 'traefik.http.routers.spring3hibernate-
   canary.rule=Host(`uat-spring.opstree.com`)' \
   --label "traefik.port=8080" --label "traefik.weight=10" --
   label "traefik.backend=app_weighted" \
   opstree/spring3hibernate:${CANARY_VERSION}
   sudo docker run -itd --name spring3hibernate-baseline --
   label traefik.enable=true \
   --label 'traefik.http.routers.spring3hibernate-
   baseline.rule=Host(`uat-spring.opstree.com`)' \
   --label "traefik.port=8080" --label "traefik.weight=90" --
   label "traefik.backend=app_weighted" \
   opstree/spring3hibernate:${BASELINE_VERSION}
   """
}
stage("Validation for Application") {
   returnValue = input message: 'Do you want to proceed
   further?',
   parameters: [choice(choices: ['Rollback', 'Proceed'], name:
   'action')]
}
if ("${returnValue}" == "Rollback") {
   stage("Rollback complete traffic to Baseline") {
    sh """
    sudo docker rm -f spring3hibernate-baseline
    sudo docker run -itd --name spring3hibernate-baseline --
    label traefik.enable=true \
    --label 'traefik.http.routers.spring3hibernate-
    baseline.rule=Host(`uat-spring.opstree.com`)' \
    --label "traefik.port=8080" --label "traefik.weight=100" --
    label "traefik.backend=app_weighted" \
    opstree/spring3hibernate:${BASELINE_VERSION}
    sudo docker rm -f spring3hibernate-canary
    """
   }
} else if ("${returnValue}" == "Proceed") {
   stage("Shifting all traffic to canary") {
    sh """
    sudo docker rm -f spring3hibernate-canary
    sudo docker run -itd --name spring3hibernate-canary --label
    traefik.enable=true \
```

```
        --label 'traefik.http.routers.spring3hibernate-
        baseline.rule=Host(`uat-spring.opstree.com`)' \
        --label "traefik.port=8080" --label "traefik.weight=100" --
        label "traefik.backend=app_weighted" \
        opstree/spring3hibernate:${CANARY_VERSION}
        """
      }
  }
}
```

Code 8.6

Continuous Deployment for Production Environment (Canary Deployment)

Figure 8.17 shows the architecture diagram of Canary deployment for production environment:

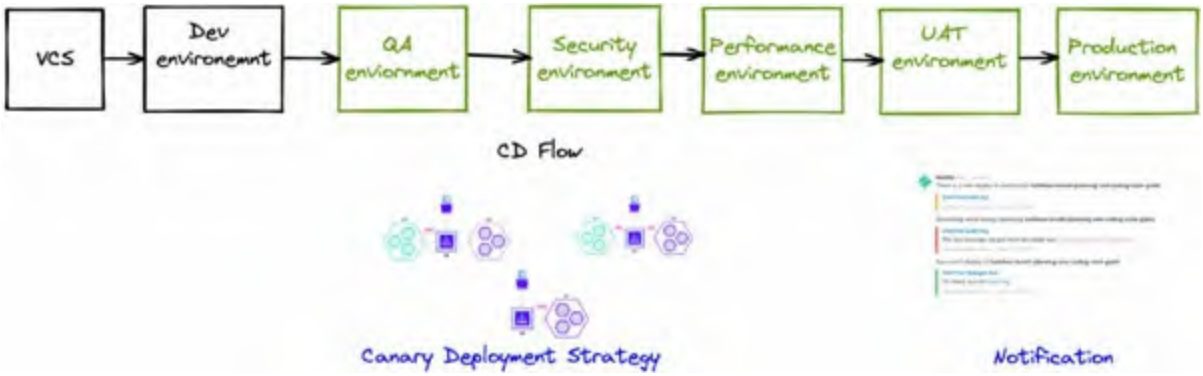

Figure 8.17: *Flow diagram for Production Deployment*

The production environment is similar to the UAT environment, so the deployment strategy will also be the same for this environment. As discussed in Sprint planning, the production environment will also use the Canary deployment for the application release. So, all the pipeline and flow will be the same for it.

Reflection

A few months after the implementation, everybody met in the break room for their start-of-the-day refreshments and casually talked about their experiences with the new setup. Sajal recalled, "The other day, I was speaking to Vishant. He was talking about the vacation he has been planning and stuff. But I found

it interesting that he mentioned in the passing that it had been quite a productive month, and that he deserves a break." "Haha," chuckled Adeel, "Yes, he has been working quite hard, but that does not sound like something he would say." I agreed, saying, "Yeah, I remember talking to him in the initial days. He was one sad soul, envious of gardeners and whatnot." Everyone burst into laughter with that recollection, which surprised me as I was unaware that he did it often. When the laughter died, Sonia added, "My team has a similar experience. With the CI/CD pipeline setup, everything is smoother. Vishant has been meeting our timelines. We have ample time to try out new things, and with a couple of new hires, we can focus on setting up our new automated testing frameworks." "Hey, Sonia, I'd like to come by later to inquire about that, if you don't mind," said Abhishek. "Sure thing, no problem, boss," she replied.

Adeel followed up on Sonia's thought, "I have to agree with Sonia there. This gradual adoption of DevOps has been quite fruitful. I remember how a lot of time was wasted on issues that had nothing to do with the actual service we were trying to roll out. Development, QA, and production were mixed, and there were no clear boundaries. It took time even to trace the point of origin of the issue. Now, it is easier to trace the origin, but we also do not need to do that, as most issues get caught in their respective environments." Sandeep added, "For me, the best part has been deployments since CI/CD was set up. I remember how Adeel, Sonia, and their teams would band their heads, trying to figure out what was breaking. Now, the releases are on time, and the respective teams can also focus on improving the quality of our services." I remember feeling a sense of comfort about this conversation and said, "Well, guys, there's still a lot of room to improve. Now that one part is completed and running smoothly, I am excited about the next steps."

Conclusion

Going through this story was quite a ride, especially when the end is not as one expected. But when adopting something new, we cannot just rely on one level and its outcome. It is only natural to seek out additional levels. Even if the original story is believable, reading more will only strengthen our decision and bring more confidence. Before it starts to sound too vague, let me clear up that in the next chapter, where we'll use case studies. It will involve an understanding and demonstration of the CI process for different

types of languages. In the end, there’s also a surprise: a peak into our very own CI/CD product.

Index

A

B

C

D

E

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

Z