Compliments of Red Hat Open Innovation Labs

DevOps Culture and Practice with OpenShift

Deliver continuous business value through people, processes, and technology

Section 6: Build It, Run It, Own It

DevOps Culture and Practice with OpenShift

Deliver continuous business value through people, processes, and technology

Tim Beattie

Mike Hepburn

Noel O'Connor

Donal Spring

BIRMINGHAM—MUMBAI

Welcome to DevOps Culture and Practice with OpenShift.

This book will enable readers to learn, understand, and apply many different practices - some people-related, some process-related, some technology-related - to make DevOps adoption and in turn OpenShift a success within their organization. It introduces many DevOps concepts and tools that we use to connect DevOps culture and practices through a continuous loop of discovery, pivots and delivery. All of this is underpinned by a foundation of culture, collaboration, and engineering.

This book provides an atlas demonstrating how to build empowered product teams within your organisation. Through a combination of real world stories, a fabricated use case (especially fun for dog and cat lovers), facilitation guides and the technical details of how to implement it all, this book provides tools and techniques to build a DevOps culture within your organization on Red Hat's OpenShift Container Platform.

It's a collection of agile, lean, design thinking, DevOps, culture, facilitation and hands-on technical enablement books all in one! As Gabrielle Benefield (Business Outcomes thought-leader) explains in her foreword, this book is "like having a great travel guide you can pull out on your journey, that gives you the direction and ideas you need when you need them. I use this book as a go-to reference that I can give to teams to help them get up and running fast. I also love that the authors speak with candor and share their real-world war stories including the mistakes and pitfalls."

To help navigate round the book, the 18 chapters have been organized into 7 sections:

- **Section 1**, Practice makes perfect introduces DevOps culture and practices. It'll also give an overview of the navigator we will use to work our way round how we use continuous discovery and continuous delivery to achieve DevOps culture.
- **Section 2**, Establishing the foundation provides practices we use to establish an open culture enabling high performing teams to realize DevOps and the technical foundation practices they use to bootstrap to achieve DevOps.
- **Section 3**, Discover It explains practices we use to discover why, for who and how we build great application products to run on OpenShift and deliver early and continuous business value.
- **Section 4**, Prioritize It shows how we decide what to work on by taking an experimental approach and how we organize our work according to business value and risk.

- **Section 5**, Deliver It covers Agile and waterfall delivery approaches and the techniques we use to measure and learn, at several levels, from iterative and incremental delivery.

- **Section 6**, Built It, Run It, Own It looks at the technology and walks through the steps, patterns and tools we use to confidently deliver and operate our case study application and platform.

- **Section 7**, Improve It, Sustain It describes how we continue round the infinite loop to continuously learn about and improve our products and technology and how the same mental model used for application products can be applied to platforms and strategy.

Table of Contents

Preface iii

Section 6: Build It, Run It, Own It 527

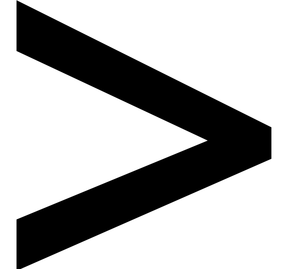
Chapter 14: Build It 531

Cluster Resources 533
Existing PetBattle Architecture 533
    PetBattle Components 535
    Plan of Attack 537
Running PetBattle 538
Argo CD 543
Trunk-Based Development and Environments 545
The Anatomy of the App-of-Apps Pattern 546
Build It – CI/CD for PetBattle 549
    The Big Picture 549
    The Build 551
    The Bake 552
    The Deploy 553
    System Test 554
    Promote 555
    Choose Your Own Adventure 558
Jenkins–The Frontend 559
    Connect Argo CD to Git 559
    Secrets in Our Pipeline 563
    The Anatomy of a Jenkinsfile 572

Branching 579
Webhooks 580
Jenkins 581
Bringing It All Together 581
What's Next for Jenkinsfile 584
Tekton–The Backend 585
Tekton Basics 585
Reusable Pipelines 588
Build, Bake, Deploy with Tekton 589
Triggers and Webhooks 593
GitOps our Pipelines 595
Which One Should I Use? 596
Conclusion 598
Chapter 15: Run It 599
The Not Safe For Families (NSFF) Component 600
Why Serverless? 600
Generating or Obtaining a Pre-trained Model 601
The OpenShift Serverless Operator 603
Deploying Knative Serving Services 604
Invoking the NSFF Component 607
Let's Talk about Testing 611
Unit Testing with JUnit 612
Service and Component Testing with REST Assured and Jest 613
Service Testing with Testcontainers 616
End-to-End Testing 617
Pipelines and Quality Gates (Non-functionals) 621
SonarQube 621

Perf Testing (Non-Functional) 627
Resource Validation 633
Image Scanning 636
Linting 638
Code Coverage 639
Untested Software Watermark 642
The OWASP Zed Attack Proxy (ZAP) 643
Chaos Engineering 644
Accidental Chaos Testing 646
Advanced Deployments 648
A/B Testing 649
The Experiment 649
Matomo – Open Source Analytics 650
Deploying the A/B Test 652
Understanding the results 655
Blue/Green deployments 656
Deployment previews 658
Conclusion 660
Chapter 16: Own It 661
Observability 661
Probes 662
Domino Effect 664
Fault Tolerance 664
Logging 665
Tracing 666
Metrics 666
Configuring Prometheus To Retrieve Metrics From the Application 669

Visualizing the Metrics in OpenShift 671
Querying using Prometheus 671
Visualizing Metrics Using Grafana 672
Metadata and Traceability 673
Labels 673
Software Traceability 676
Annotations 677
Build Information 677
Alerting 678
What Is an Alert? 678
Why Alert? 678
Alert Types 679
Managing Alerts 680
User-Defined Alerts 680
OpenShift Alertmanager 684
Service Mesh 685
Why Service Mesh? 685
Aside – Sidecar Containers 686
Here Be Dragons! 687
Service Mesh Components 688
PetBattle Service Mesh Resources 689
Operators Everywhere 693
Operators Under the Hood 695
Control Loops 695
Operator Scopes 696
Operators in PetBattle 697
Service Serving Certificate Secrets 700
Conclusion 701



Preface

About

This section briefly introduces the authors, the coverage of this book, the skills you'll need to get started, and the hardware and software needed to complete all of the technical topics.

About DevOps Culture and Practice with OpenShift

DevOps Culture and Practice with OpenShift features many different real-world practices - some people-related, some process-related, some technology-related - to facilitate successful DevOps, and in turn OpenShift, adoption within your organization. It introduces many DevOps concepts and tools to connect culture and practice through a continuous loop of discovery, pivots, and delivery underpinned by a foundation of collaboration and software engineering.

Containers and container-centric application lifecycle management are now an industry standard, and OpenShift has a leading position in a flourishing market of enterprise Kubernetes-based product offerings. *DevOps Culture and Practice with OpenShift* provides a roadmap for building empowered product teams within your organization.

This guide brings together lean, agile, design thinking, DevOps, culture, facilitation, and hands-on technical enablement all in one book. Through a combination of real-world stories, a practical case study, facilitation guides, and technical implementation details, *DevOps Culture and Practice with OpenShift* provides tools and techniques to build a DevOps culture within your organization on Red Hat's OpenShift Container Platform.

About the authors

Tim Beattie is Global Head of Product and a Senior Principal Engagement Lead for Red Hat Open Innovation Labs. His career in product delivery spans 20 years as an agile and lean transformation coach - a continuous delivery & design thinking advocate who brings people together to build meaningful products and services whilst transitioning larger corporations towards business agility. He lives in Winchester, UK, with his wife and dog, Gerrard the Labrador (the other Lab in his life) having adapted from being a cat-person to a dog-person in his 30s.

Mike Hepburn is Global Principal Architect for Red Hat Open Innovation Labs and helps customers transform their ways of working. He spends most of his working day helping customers and teams transform the way they deliver applications to production with OpenShift. He co-authored the book "DevOps with OpenShift" and loves the outdoors, family, friends, good coffee, and good beer. Mike loves most animals, not the big hairy spiders (Huntsman) found in Australia, and is generally a cat person unless it's Tuesday, when he is a dog person.

Noel O'Connor is a Senior Principal Architect in Red Hat's EMEA Solutions Practice specializing in cloud native application and integration architectures. He has worked with many of Red Hat's global enterprise customers in both Europe, Middle East & Asia. He co-authored the book "DevOps with OpenShift" and he constantly tries to learn new things to varying degrees of success. Noel prefers dogs over cats but got overruled by the rest of the team.

Donal Spring is a Senior Architect for Red Hat Open Innovation Labs. He works in the delivery teams with his sleeves rolled up tackling anything that's needed - from coaching and mentoring the team members, setting the technical direction, to coding and writing tests. He loves technology and getting his hands dirty exploring new tech, frameworks, and patterns. He can often be found on weekends coding away on personal projects and automating all the things. Cats or Dogs? He likes both :)

About the illustrator

Ilaria Doria is an Engagement Lead and Principal at Red Hat Open Innovation Labs. In 2013, she entered into the Agile arena becoming a coach and enabling large customers in their digital transformation journey. Her background is in end-user experience and consultancy using open practices to lead complex transformation and scaling agile in large organizations. Colorful sticky notes and doodles have always been a part of her life, and this is why she provided all illustrations in the book and built all digital templates. She is definitely a dog person.

About the reviewer

Ben Silverman is currently the Chief Architect for the Global Accounts team at Cincinnati Bell Technology Services. He is also the co-author of the books *OpenStack for Architects*, *Mastering OpenStack*, *OpenStack – Design and Implement Cloud Infrastructure*, and was the Technical Reviewer for *Learning OpenStack* (Packt Publishing).

When Ben is not writing books he is active on the Open Infrastructure Superuser Editorial Board and has been a technical contributor to the Open Infrastructure Foundation Documentation Team (Architecture Guide). He also leads the Phoenix, Arizona Open Infrastructure User Group. Ben is often invited to speak about cloud and Kubernetes adoption, implementation, migration, and cultural impact at client events, meetups, and special vendor sessions.

Learning Objectives

- Implement successful DevOps practices and in turn OpenShift within your organization
- Deal with segregation of duties in a continuous delivery world
- Understand automation and its significance through an application-centric view
- Manage continuous deployment strategies, such as A/B, rolling, canary, and blue-green
- Leverage OpenShift's Jenkins capability to execute continuous integration pipelines
- Manage and separate configuration from static runtime software
- Master communication and collaboration enabling delivery of superior software products at scale through continuous discovery and continuous delivery

Audience

This book is for anyone with an interest in DevOps practices with OpenShift or other Kubernetes platforms.

This DevOps book gives software architects, developers, and infra-ops engineers a practical understanding of OpenShift, how to use it efficiently for the effective deployment of application architectures, and how to collaborate with users and stakeholders to deliver business-impacting outcomes.

Approach

This book blends to-the-point theoretical explanations with real-world examples to enable you to develop your skills as a DevOps practitioner or advocate.

Hardware and software requirements

There are five chapters that dive deeper into technology. *Chapter 6*, *Open Technical Practices - Beginnings, Starting Right* and *Chapter 7*, *Open Technical Practices - The Midpoint* focuses on boot-strapping the technical environment. *Chapter 14*, *Build It*, *Chapter 15*, *Run It*, and *Chapter 16*, *Own It* cover the development and operations of features into our application running on the OpenShift platform.

We recommend all readers, regardless of their technical skill, explore the concepts explained in these chapters. Optionally, you may wish to try some of the technical practices yourself. These chapters provide guidance in how to do that.

The OpenShift Sizing requirements for running these exercises are outlined in Appendix A.

Conventions

Code words in the text, database names, folder names, filenames, and file extensions are shown as follows:

We are going to cover the basics of component testing the PetBattle user interface using Jest. The user interface is made of several components. The first one you see when landing on the application is the home page. For the home page component, the test class is called `home.component.spec.ts`:

```
describe('HomeComponent', () => {
  let component: HomeComponent;
  let fixture: ComponentFixture<HomeComponent>;

  beforeEach(async () => {...
  });

  beforeEach(() => {...
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });
});
```

Downloading resources

All of the technology artifacts are available in this book's GitHub repository at https://github.com/PacktPublishing/DevOps-Culture-and-Practice-with-OpenShift/

High resolution versions of all of the visuals including photographs, diagrams and digital artifact templates used are available at https://github.com/PacktPublishing/DevOps-Culture-and-Practice-with-OpenShift/tree/master/figures

We also have other code bundles from our rich catalog of books and videos available at https://github.com/PacktPublishing/. Check them out!

We are aware that technology will chage over time and APIs will evolve. For the latest changes of technical content, have a look at the book's GitHub repository above. If you want to contact us directly for any issue you've encountered, please raise an issue in this repository.

Section 6: Build It, Run It, Own It

In previous sections, we've been discussing the approach we're taking in discovering and prioritizing work to deliver applications such as PetBattle. This includes the many aspects we need to consider when building the solution's components. Now it's time to actually deliver working software:

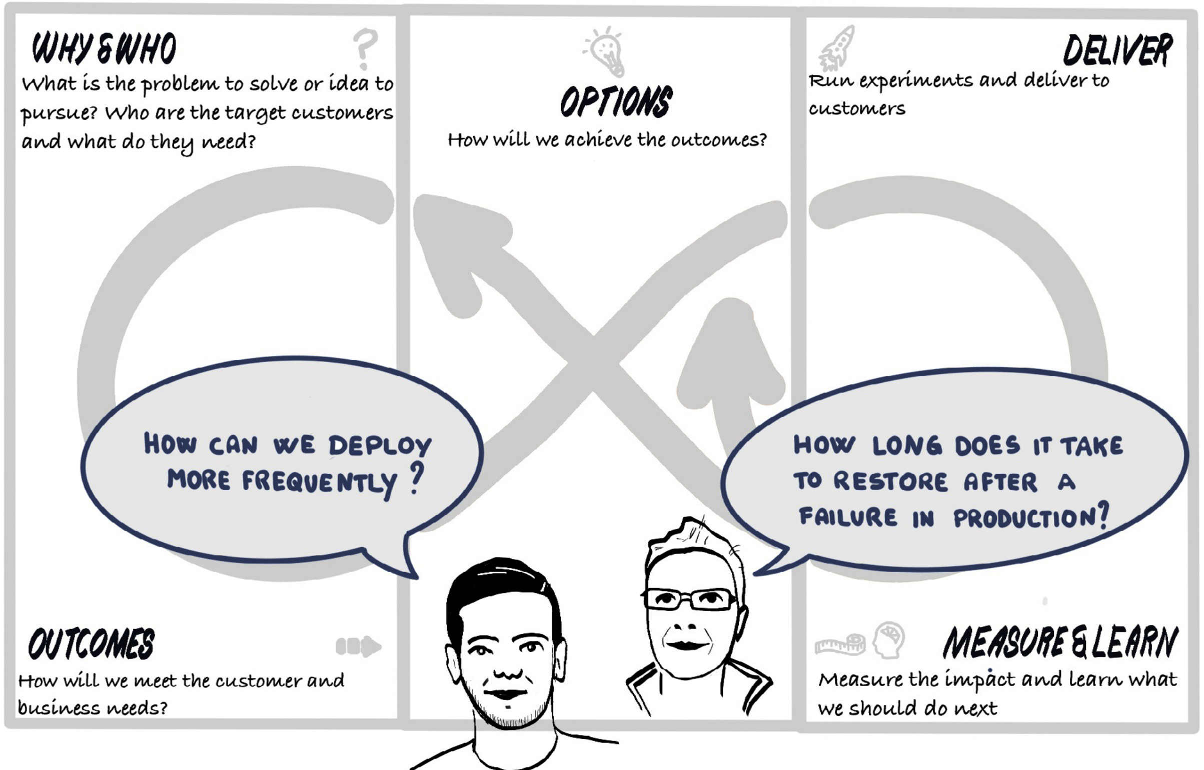


Figure 14.0.1: Focusing on the How

This section is where we do the following:

- Build the solution components (the API, frontend, and so on) using our preferred automation stack.
- Deploy these components onto the OpenShift Container Platform in a repeatable way.
- Own and manage them so that we can watch the site grow and thrive in a culture that empowers our community of motivated developers.

Once our master cat plan is complete, we will become filthy rich in the process! As PetBattle investors, we can then buy islands, invest in space engineering, or just sip martinis by the pool all day until it becomes so boring that we start a new cryptocurrency named *Pet-Coin* and lose it all.

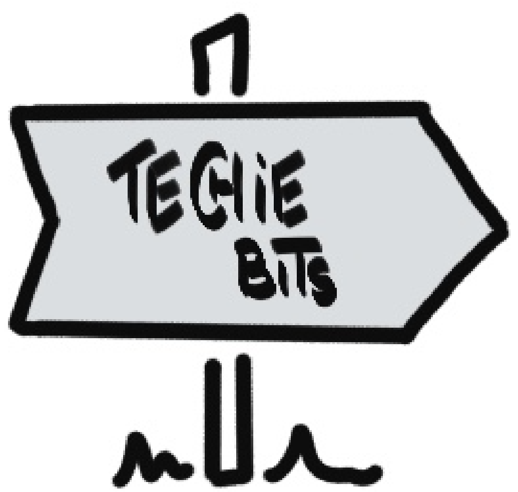
To set expectations correctly, we're not really going into detail about all of the application code itself. All the source code is available (in the Git repositories for the book) and you can go through it at your own pace. There is a lot of value going through the code, in particular the Helm, Quarkus, and AI code.

Our focus is on how to use the tooling provided within OpenShift to build, deploy, and run the applications. We will go into certain advanced features, such as Operators, Serverless, Service Mesh, and CI/CD tooling examples and techniques. Our intention is to provide examples and advice around why we choose to use certain tools or techniques and approaches and how all of them fit together in a cohesive approach specific to PetBattle. You can then pick and choose which ones you want to use in your own projects.

The section is broken down into three parts:

1. *Chapter 14*, *Build It*: This is where we introduce how we will use Git as the single source of truth. We will cover taking our source code and packaging it using either Tekton or Jenkins.
2. *Chapter 15*, *Run It*: This section covers testing, introducing a new component to our app with Knative, running A/B tests, and capturing user metrics using some of the advanced deployment capabilities within OpenShift.
3. *Chapter 16*, *Own It*: This section covers keeping the lights on with monitoring and alerting feedback loops. We will also touch upon the importance of Operators in Kubernetes.

Throughout these chapters, we cover a core belief that all teams should want to take pride in their code. A team member should want to Build It, Run It, and ultimately Own It.



If you feel like the low-level technical details are a bit too much for you or not something you're super interested in, that's OK! The next three chapters may seem like a change in gear, but we'd recommend against skipping over the chapters completely.

Don't worry too much about the low-level details, code snippets, and screenshots, it's important for everyone to grasp the value of the concepts in this book. That includes the value of any technical improvements the team wants to make. The following chapters should help articulate that.

14

Build It

"*It works on my machine*"—a phrase heard time and time again by developers, testers, and operators as they write, test, and verify their code. *It works on my machine* is a phrase rooted in siloed teams where ownership of the problem moves around like a tennis ball on a court at Wimbledon. The metaphorical wall that exists between teams that have operated in silos, passing work over the wall and not taking full responsibility for the end-to-end journey is a problem that has been around for decades. We need to break away from this behavior! From now on, it's not, "*It's working on my machine*" but rather, "*How has your code progressed in the build system*?" The build and deployment pipeline our code runs through is a shared responsibility. In order for this to be the case, all team members must contribute to the pipeline and be ready to fix it when it breaks.

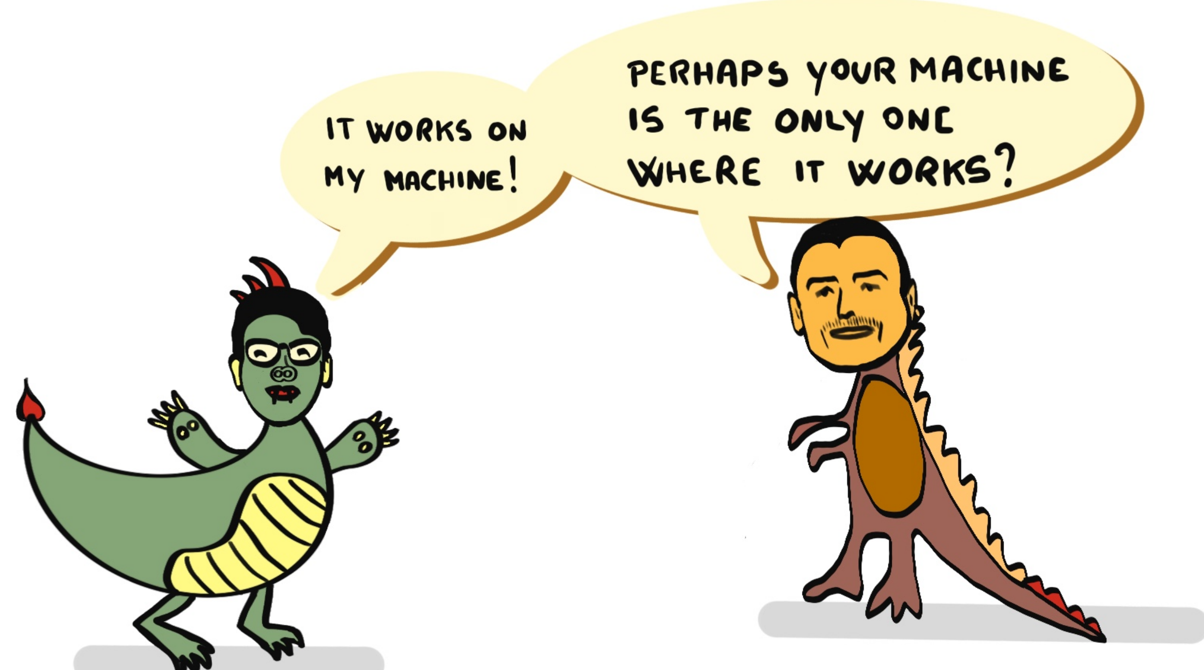


Figure 14.1: It works on my machine

If your code has failed the build because you forgot to check in a dependency, or if your tests are not passing, then it is your responsibility to fix it! The purpose of the deployment pipeline is to create a repeatable process that our code will pass through and accelerate releases while also de-risking them. If we know that on every commit to a repository all of our tests are executed, we're going to have a large amount of confidence that the resulting application will work well. If we're continuously increasing the test volume as the application's complexity increases, that too should grow our confidence. It's critical for teams to want to own their software pipelines.

Having a platform such as OpenShift is a bit like the Beatles singing on the rooftop of the Apple Corps building on Saville Row about people *coming together*. Developers, operations, testers, architects, designers, database administrators, and analysts, everyone coming together and using a platform like OpenShift provides a shared space to collaborate upon. Building applications and business services on the platform where developers can self-service all of their requirements in a safe, access-controlled manner, bringing down the walls between teams, and removing bottlenecks such as having to wait for permissions to deploy an application—this gets everyone speaking the same language to deliver business outcomes through modern application delivery and technological solutions.

Cluster Resources

This section of the book will be one of the most technical. As described in the *Appendix*, the minimum requirements for running the code examples using **CodeReady Containers** (**CRCs**) in this chapter are as follows:

Scenario	CRC Command	Comments
Chapter 14, Build It	`crc start -c 4 -m 16384 -d 50`	Argo CD bootstrap, tool install, application install, CRCs, 4 cores, 16 GB RAM, 50 GB disk

Table 14.1: Minimum requirements for running code examples using CRCs

With the amount of memory required to follow, and the technical content out of the way, let's dive into things in more detail. We'll start by looking over the components of the existing PetBattle applications as they move from a hobby weekend project into a highly available, production-based setup that is built and maintained by a strong, cross-functional team.

Existing PetBattle Architecture

The initial PetBattle architecture was pretty basic and revolved around deploying application components running in a single **virtual machine** (**VM**). The initial architecture had three major components: a JavaScript frontend; a Java-based backend, providing an API and a database; and a single instance of MongoDB. Nothing here was too exciting or complex but hidden inside was a minefield of technical debt and poor implementation that caused all sorts of problems when the site usage took off.

The issues with this architecture seemed to include:

- A monolith—Everything had to be deployed and scaled as one unit, there were no independent moving parts.
- Authentication and access control were non-existent.
- Tests? Unit tests? But seriously, there weren't many.
- It required a lot of data maintenance as everything was stored in the database.
- Bad actors adding inappropriate images to our family-friendly cat application.
- Fragile application—If something went wrong, the application would crash and everything had to be restarted.

Back in *Chapter 9*, *Discovering the How*, we went through an Event Storming exercise that helped drive a newly proposed architecture. It consisted of a UI component and a backing service that provided different REST-based APIs to the UI, as shown in *Figure 14.2*:

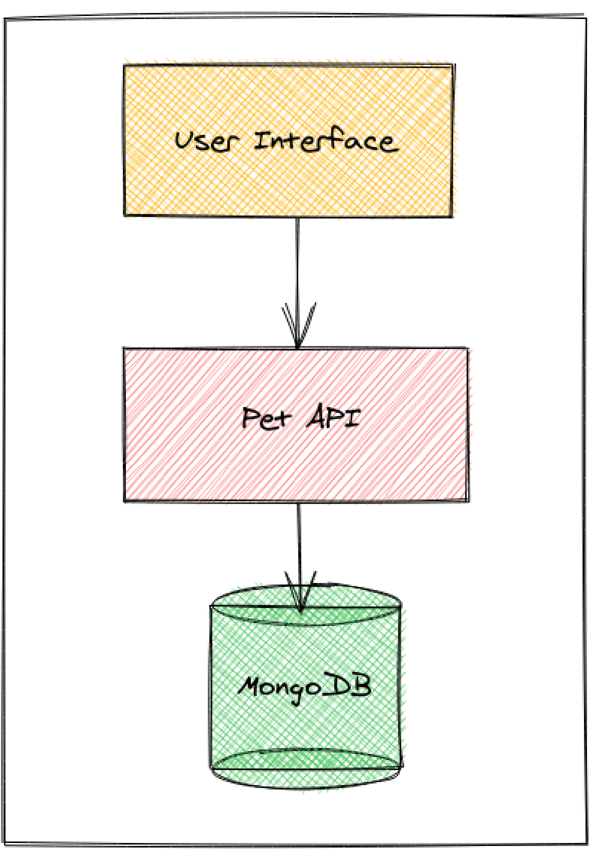


Figure 14.2: PetBattle's initial hobbyist architecture

Let's now take a look at the individual PetBattle components.

PetBattle Components

In the coming chapters, we will explore automation, testing, and the extension of PetBattle to include aspects such as monitoring and alerting, Knative Serving, and Service Mesh. But first, let's imagine that the PetBattle team has completed a few sprints of development. They have been building assets from the Event Storms and now have the components and architecture as seen in *Figure* 14.3. Through the Event Storm, we also identified a need for authentication to manage users. The tool of choice for that aspect was Keycloak.

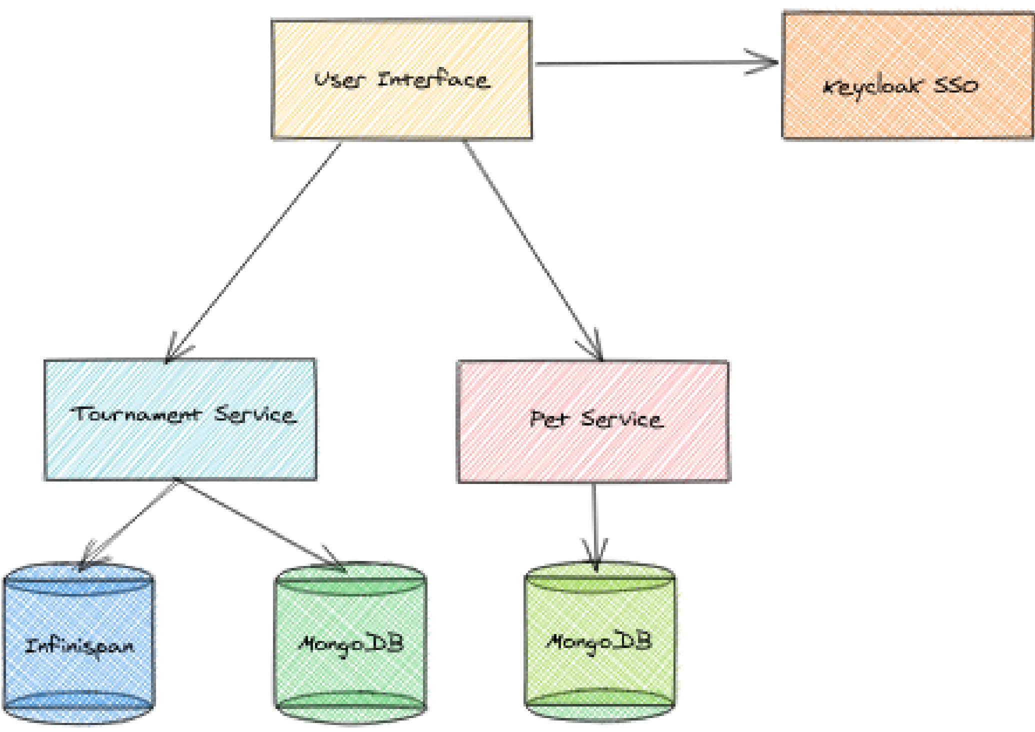


Figure 14.3: PetBattle's evolving architecture

PetBattle's architecture is now increasing in complexity. It has a UI that connects to two services to provide it with data. **Single Sign-On** (**SSO**) and user management are provided by Keycloak. Let's take a look at each component of the architecture in more detail.

User Interface

The UI is written in Angular[1] v12, a complete JavaScript framework from Google for building web and mobile applications. The application is transpiled and the static site code is then served from a container running Nginx (a webserver) instance provided by Red Hat. The application is set up to pull its configuration on startup, which sets up endpoints for all of the dependent services, such as Keycloak and the APIs. This configuration is managed as a ConfigMap in OpenShift.

Pet Service

The Pet service is a straightforward service that uses Java Quarkus[2] as the framework, backed by a MongoDB database to retrieve and store details of the pets uploaded to partake in a tournament.

Tournament Service

The Tournament service also uses the Quarkus framework and stores the state in both MongoDB and an Infinispan distributed cache. MongoDB is used to store the details of the tournament such as which pet won the tournament—but why did we use a cache?

Well, the answer is that a tournament only exists for a finite period of time and using a database to store temporal data is not a great fit for our use case. Also, Infinispan stores the cache data in memory, which is much faster to access than data on disk. The drawback of this is that if the Infinispan pod dies/crashes, then the data is lost. However, we plan to circumvent this in production by having at least two replicas, with the data being replicated between the pods.

User Management

User management, authentication, and access control are a few other critical parts of the architecture that need to be addressed. We're using Keycloak,[3] an open source identity and access management tool, to provide this functionality. We could have written some code ourselves for this functionality, but security is an area that requires a lot of expertise to get it right, and Keycloak does a great job of using open standards to do this job correctly.

---

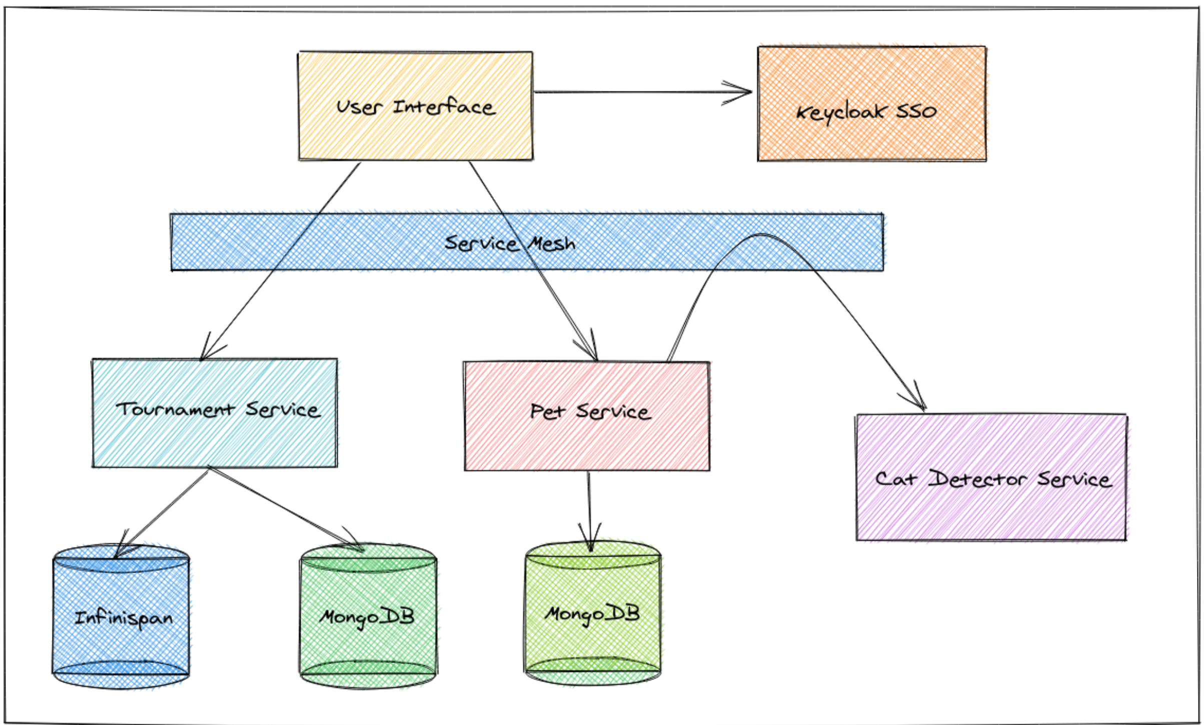
1 https://angular.io/

2 https://quarkus.io/

3 https://www.keycloak.org/

Plan of Attack

Initially, we are going to get the core PetBattle application components and services up and running on OpenShift in a fairly manual way. We want to be able to develop locally, adding new functionality to show how easy it is to combine Helm and OpenShift to repeatedly deploy our code. Once that is completed, we are going to automate the setup and deployment process using various tools, including Tekton/Jenkins, Argo CD, and GitOps. We will explore how to add new components to our architecture using Knative and experiment with some of the more advanced deployment capabilities that we can utilize. Finally, in *Chapter 16*, *Own It*, we will look at application monitoring and alerting along with Service Mesh for traceability. *Figure 14.4* shows the additional components added to the architecture, including the Knative Cat Detector Service being proxied via the Service Mesh.



Final OpenShift Deployment

Figure 14.4: PetBattle's target architecture, including final OpenShift deployment

We will be using the command line as much as possible to show and explain the commands involved. Each step can also be performed via the OpenShift web console. If you're new to OpenShift, the web console is a great place to get started as it's full of tips and tutorials!

Running PetBattle

In *Chapter 6*, *Open Technical Practices – Beginnings*, *Starting Right*, we talked about Helm and its use as an application lifecycle manager for installing, upgrading, and rolling back application deployments. We are going to start with the command line, but you can skip to the end of this section if you would like to follow the web console method. If you need help installing the Helm command-line tool, take a look at *Chapter 6* as a refresher. Now let's see how we can easily deploy the PetBattle suite of applications as Helm charts into a single project on OpenShift. On your terminal, add the PetBattle Helm repositories:

```
$ helm repo add petbattle \
     https://petbattle.github.io/helm-charts
```

There are three main applications that make up PetBattle and are searchable in the Helm repository:

Name	Description
`pet-battle`	PetBattle frontend – Angular app deployed on Nginx
`pet-battle-api`	PetBattle API – Cats API that stores our uploaded images in MongoDB
`pet-battle-tournament`	PetBattle Tournament – Service for managing and running each weekly competition

Table 14.2: The three main applications making up PetBattle

The infrastructure Helm chart is normally deployed as a dependency of the Tournament Helm chart but can optionally be deployed alone. This can be useful for debugging purposes. The **Not Safe For Families** (**NSFF**) component is an optional chart, adding a feature whereby the API checks uploaded images for safe content for our family-friendly application.

Name	Description
`pet-battle-infra`	PetBattle infrastructure – Chart contains Keycloak, Grafana, and alerting components
`pet-battle-nsff`	PetBattle NSFF components; includes the machine learning algorithm and APIs

Table 14.3: The infrastructure and NSFF Helm charts

We can search for the latest versions of these charts using the following command:

```
$ helm search repo pet-battle
```

Let's now deploy the main PetBattle application into our OpenShift cluster. We need to update a local copy of the PetBattle frontend's Helm `values.yaml` file to match our cluster URLs. This is needed to connect the frontend when deployed to the correct collection of backend services. We can provide these values to our Helm charts when deploying the suite of PetBattle applications. Let's download an example of the `values.yaml` file for us to edit:

```
$ wget https://raw.githubusercontent.com/petbattle/pet-battle/master/chart/
values.yaml/tmp/values.yaml
```

Open the `values.yaml` file and replace the five URLs listed in the `config_map` to match your OpenShift cluster (change the `apps.cluster.com` domain to `apps-crc.testing`, for example, if you are using a CRC). For example:

```
# custom end point injected by config map
config_map: '{
  "catsUrl": "https://pet-battle-api-petbattle.apps.cluster.com",
  "tournamentsUrl": "https://pet-battle-tournament-petbattle.apps.cluster.
com",
  "matomoUrl": "https://matomo-labs-ci-cd.apps.cluster.com/",
  "keycloak": {
    "url": "https://keycloak-petbattle.apps.cluster.com/auth/",
    "realm": "pbrealm",
    "clientId": "pbclient",
    "redirectUri": "https://pet-battle-petbattle.apps.cluster.com/*",
    "enableLogging": true
   }
}'
```

Gather the **latest chart version** for each of the PetBattle applications from the preceding Helm search command and install the three applications `pet-battle`, `pet-battle-api`, and `pet-battle-tournament` into your cluster. To do this, you will need to be logged in to your OpenShift cluster. For example:

```
# Login to OpenShift
$ oc login -u <username> --server=<server api url>

$ helm upgrade --install pet-battle-api \
petbattle/pet-battle-api --version=1.0.15 \
--namespace petbattle --create-namespace

$ helm upgrade --install pet-battle \
petbattle/pet-battle --version=1.0.6 \
-f /tmp/values.yaml --namespace petbattle

$ helm upgrade --install pet-battle-tournament \
petbattle/pet-battle-tournament --version=1.0.39 \
--set pet-battle-infra.install_cert_util=true \
--timeout=10m \
--namespace petbattle
```

If the `pet-battle-tournament` install times out, just run it again.

Each Helm install chart command should return a message similar to the following:

```
NAME: pet-battle-api
LAST DEPLOYED: Thu Feb 25 19:37:38 2021
NAMESPACE: petbattle
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

Using `helm list` should give you a list of the installed charts. You should see the following pods running in your `petbattle` project. An example is shown in *Figure 14.5*:

```
virt:~$ oc get pods
NAME                                   READY   STATUS      RESTARTS   AGE
grafana-deployment-7776b9b67c-wtvx6    1/1     Running     0          4m16s
grafana-operator-9ddb559ff-wp7dw       1/1     Running     0          5m31s
infinispan-0                           1/1     Running     0          4m33s
infinispan-operator-645c945c4-zj4g7    1/1     Running     0          5m13s
keycloak-0                             1/1     Running     0          4m19s
keycloak-operator-5fb789674d-fsxlr     1/1     Running     0          5m11s
keycloak-postgresql-58fdd7fbc7-97q62   1/1     Running     0          4m19s
pet-battle-1-42dkh                     1/1     Running     0          9m47s
pet-battle-1-deploy                    0/1     Completed   0          9m50s
pet-battle-api-6d7894bc97-2gdnr        1/1     Running     0          19m
pet-battle-api-mongodb-1-deploy        0/1     Completed   0          19m
pet-battle-api-mongodb-1-prg4s         1/1     Running     0          19m
pet-battle-tournament-1-7m2zp          1/1     Running     0          5m31s
pet-battle-tournament-1-deploy         0/1     Completed   0          5m35s
pet-battle-tournament-mongodb-1-66gdq  1/1     Running     0          5m35s
pet-battle-tournament-mongodb-1-deploy 0/1     Completed   0          5m38s
```

Figure 14.5: PetBattle pods

The Tournament service will take several minutes to deploy and stabilize. This is because its dependent infrastructure chart is deploying operator subscriptions for Keycloak, Infinispan, and Grafana. Navigate to the OpenShift web console and you should now be able to explore the PetBattle application suite as shown in *Figure 14.6*. Browse to the PetBattle frontend to play with the applications.

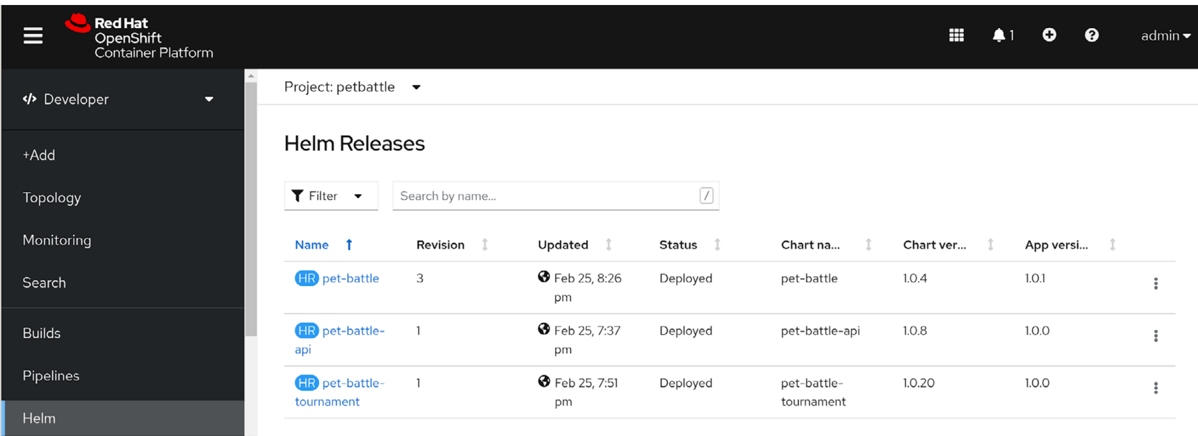


Figure 14.6: PetBattle Helm charts deployed in the OpenShift Developer view

You have now been shown how to install PetBattle Helm charts using the command line—some may say the hard way! We are now going to demonstrate some of the integrated features of Helm in OpenShift—some may say the easier way! We can create a `HelmChartRepository` Custom Resource object that points to our PetBattle Helm chart repository; think of it as `helm repo add` for OpenShift. Run this command to install the chart repository:

```
cat <<EOF | oc apply -f -
apiVersion: helm.openshift.io/v1beta1
kind: HelmChartRepository
metadata:
  name: petbattle-charts
spec:
  name: petbattle
  connectionConfig:
    url: https://petbattle.github.io/helm-charts
EOF
```

With this in place, we can browse to the Developer view in OpenShift and select Add Helm Charts, and see a menu and a form-driven approach to installing our Helm charts—just select a chart and install it:

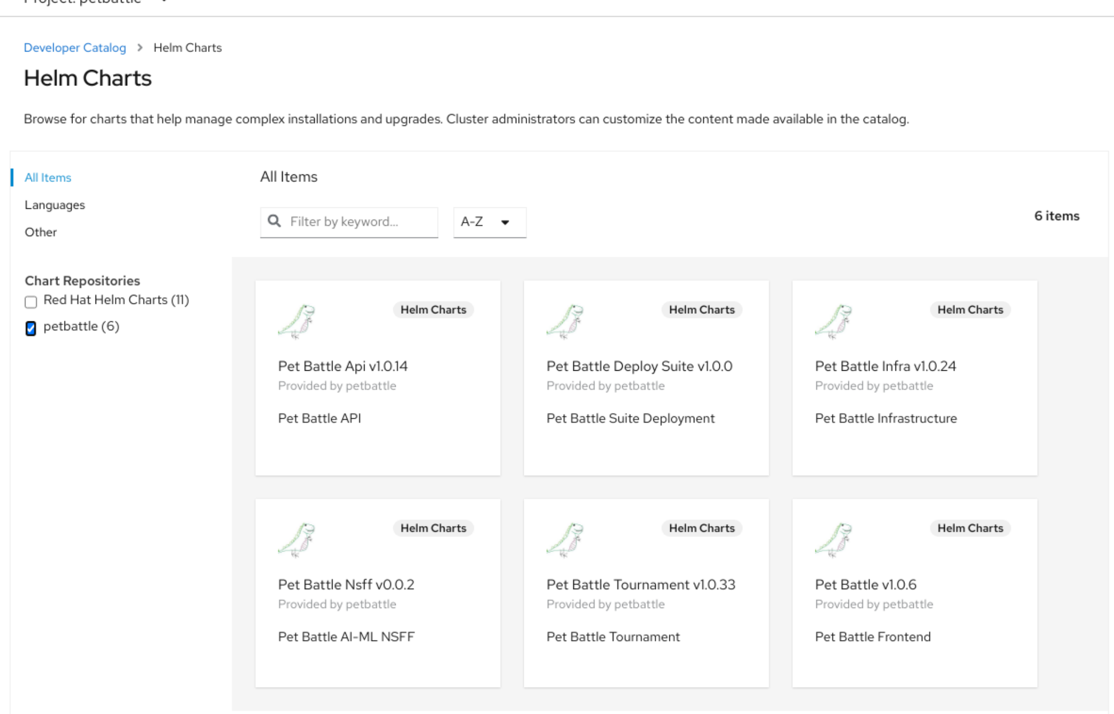


Figure 14.7: Adding PetBattle via the HelmChartRepository view in OpenShift

This can provide a great developer experience for teams sharing services with each other. A backend team can produce a new app to the repository and a downstream team can deploy it to their developer environment in a single click. In fact, if you add a Helm values schema file, OpenShift will build a **What You See Is What You Get** (**WYSIWYG**) form for easy configuration of the values file.

Argo CD

When we established our foundations in *Section 2, Establishing the Foundation*, we bootstrapped all of our builds, deployment, and tooling using Helm and Argo CD. We made some opinionated choices when running that bootstrap automation and it's worth discussing some of the trade-offs we made in a bit more detail. We followed our call to action when establishing our technical foundation and planned out what worked for us as the PetBattle product team and reviewed and discussed what was working and not working so well.

It turned out that for our development team, bootstrapping all of the CI/CD tools became an extremely important task. We had been given an arbitrary (but necessary) constraint that our development OpenShift cluster needed to be rebuilt from scratch every two weeks. So we needed to be confident that our CI and CD could stand up quickly and repeatedly. By following our everything-as-code practice, all of our OpenShift infrastructure definitions, CI/CD tooling, and pipeline definitions are stored in Git. The declared CI/CD tooling state is continuously synced to our development cluster by Argo CD, so updating the SonarQube Helm chart version, for example, is as easy as changing one line and pushing it to Git. The change is synchronized and rolled out to our cluster a minute later.

Being able to effectively lifecycle-manage all of the supporting tools involved with building your applications takes effort and attention to detail, but it is worth it in the long run, as you will have built a system that can handle change easily and repeatedly. We have optimized our application lifecycle around the cost of change, making the cost (in man hours) as small as possible. Human time is our biggest resource cost after all!

The versions of all our tooling are checked into Git using **Semantic Versioning**[4] (**SemVer**). There is a good lesson to be learned here in terms of version control; nearly all modern open-source software uses this pattern for it. Often you can be surprised with the resulting deployment when a chart or operator references the *latest* images from external sources. The *latest* tag is a moving target and is updated often. Referencing tags for your versions in Git is like walking a tightrope, there is a balancing act to be performed between wanting to easily accept your toolchain and wanting to update it for bugs and security fixes—hence the use of tags and knowing with confidence that a specific version is working. Normally in SemVer `MAJOR.MINOR` versions are tags that move with small bug fixes and security patches. `MAJOR.MINOR.PATCH` versions are not tags and yet they specify a fixed version (ideally!). Choose a strategy that does not incur too much technical debt that strands the team on old and unsupported versions forever. This is balanced with not having to constantly update version numbers all the time. Of course, if you have optimized for a small cost of change through automation, this problem of changing versions becomes much less of an issue!

We have chosen a *push* (CI) and *pull* (CD) model for our software delivery lifecycle. The job of building images and artifacts (Helm charts and configuration), as well as unit and integration testing, is part of a *push* CI model. On every code commit, a build pipeline trigger (Tekton or Jenkins) fires. It is the job of the Argo CD controller to keep what we have deployed in our OpenShift cluster in sync with the declared application state in our Git repositories. This is a GitOps pull model for CI. The key thing here is that Git is the single source of truth and everything can be recreated from this source.

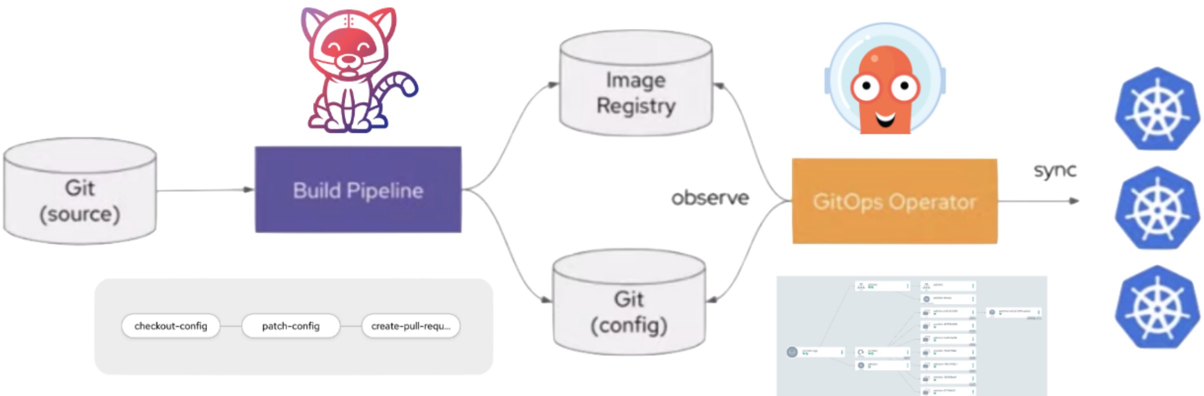


Figure 14.8: A GitOps push and pull model for continuous delivery

4 https://semver.org/

The main benefit we see with this approach is that it is developer-centric. Any change in the code base triggers a pipeline build and deployment. This gives the team fast feedback for any breakages, since automated tests are always run against the new code. The pull CD model decouples the synchronous nature of a build and testing pipeline. Built artifacts (container images and configuration) can be built once, then tagged and promoted through a lifecycle, all of which is controlled from Git. This is great for auditability and discovering who changed what and when. We can easily trace code committed and pushed with builds, tests, and deployments. It is also a flexible approach in that not all artifacts need to be built per se. Configuration can be changed and deployed using the same model. The model is also very flexible in its ability to support different development workflow models. For example, Gitflow and Trunk-based development can easily be catered for, depending on how the team chooses to work.

Trunk-Based Development and Environments

When designing our initial pipelines, we mapped out the basic build, bake, deploy, integration testing, tag, and promotion stages. In *Figure* 14.9, we can see the MultiBranchPipeline Plugin, branches, and namespaces.

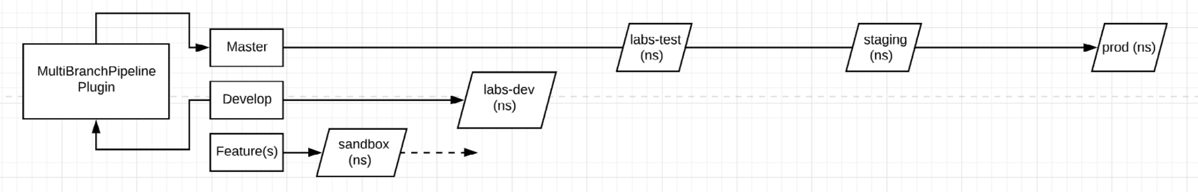


Figure 14.9: Branches and namespaces

This helped us to clarify where the responsibilities lie between our Git branches, our continuous integration tasks, continuous delivery tasks, and which OpenShift projects these would occur in.

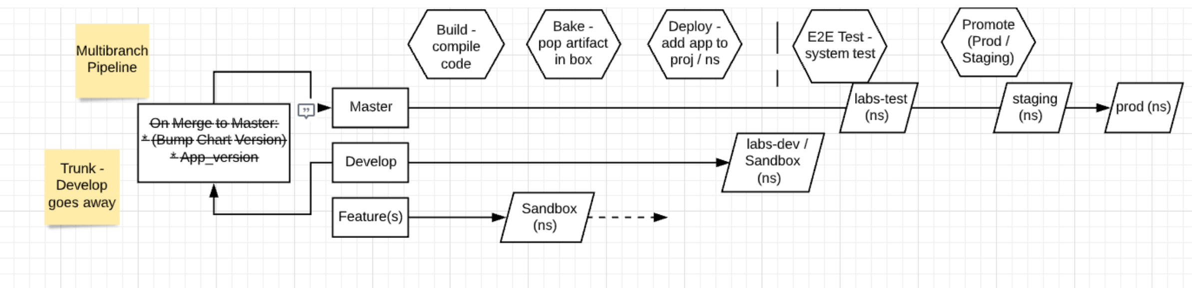


Figure 14.10: Branches and activities modeling

Because we are following trunk-based development,[5] our main/master branch is built, tagged, and promoted through the full lifecycle, that is, images are built, unit and functionally tested, and deployed into `labs-test` with end-to-end testing prior to deployment within the `labs-staging` project. For any short-lived feature branches or pull requests, we decided to only unit test, build, and deploy these sources into our `labs-dev` OpenShift project. That way we can customize which pipeline tasks happen on specific code branches. There is a trade-off between the time and resources used for every code commit in our pipelines, and this must be adjusted according to what goes into our pipelines to help improve the overall product quality.

The Anatomy of the App-of-Apps Pattern

We choose to use Helm; remember, at its most basic, Helm is just templating language for packaging our Kubernetes-based application resources. Each PetBattle application has its own Git repository and Helm chart, making it easier to code independently of other apps. This inner *Helm chart per application* box is depicted in *Figure 14.11*. A developer can get the same experience and end result installing an application chart using a `helm install` as our fully automated pipeline. This is important from a useability perspective. Argo CD has great support for all sorts of packaging formats that suit Kubernetes deployments, Kustomize, Helm, as well as just raw YAML files. Because Helm is a templating language, we can mutate the Helm chart templates and their generated Kubernetes objects with various values.

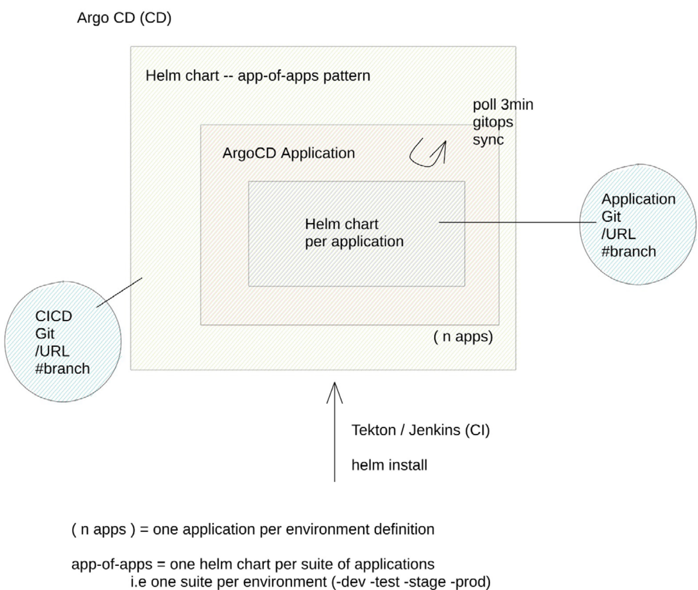


Figure 14.11: Application packaging, Helm, and Argo CD with the app-of-apps pattern

5 https://trunkbaseddevelopment.com

One strict view of GitOps is that mutating a state is not as *pure* as just checking in the filled-in templates with the values themselves. Kustomize, for example, has no templating and follows this approach. We use Kustomize for deploying our CI/CD automation with Argo CD because we think it fits that use case better. This means that we are less likely to have a large number of CI/CD environments for our PetBattle product—at the moment there is just one.

The trade-off here is that while we use GitOps to synchronize the Helm chart itself, the supply of application values may come from multiple places, so you have to be careful to understand where overriding values and precedence occurs, as follows:

- **Lowest precedence**: `values.yaml` provided with the chart (or its sub chart dependencies)—these are kept in sync by the Argo CD controller.
- **Higher precedence**: Override values specified when Argo CD creates the Helm application. These have higher precedence than value files, such as `helm template --set` on the command line. These can be specified in a template or a trigger, depending on how the pipeline is run.

We deploy each of our applications using an Argo CD application definition. We use one Argo CD application definition for every environment in which we wish to deploy the application. This is the red box depicted in *Figure 14.11*. We make use of Argo CD with the app-of-apps pattern[6] to bundle these all up; some might call this an application suite! In PetBattle we generate the app-of-apps definitions using a Helm chart. This is the third, outer green box in *Figure 14.11*. The configuration for this outer box is kept in a separate Git repository to our application.

The app-of-apps pattern is where we declaratively specify one Argo CD app that consists only of other apps. In our case, this is the `pet-battle-suite` application. We have chosen to put all of our applications that are built from the main/master under this `pet-battle-suite` umbrella. We have a PetBattle suite for *testing* and *stage* environments. *Figure 14.12* shows the app-of-apps for the stage environment:

6 https://argoproj.github.io/argo-cd/operator-manual/cluster-bootstrapping/#app-of-apps-pattern

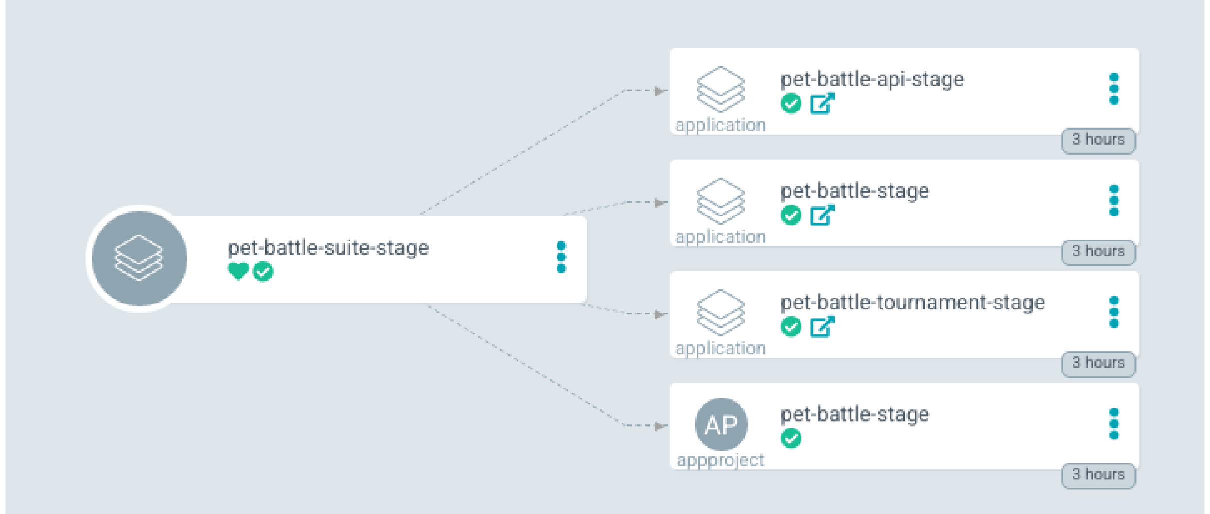


Figure 14.12: Argo CD, a deployed application suite

In Git, we model different branches using the following patterns:

- **HEAD/main/master**: An app-of-apps pattern (full suite of apps deployed and synced to all our environments, `labs-test`, `labs-staging`)
- **Branches/PRs**: A single Helm chart-deployed application in our `labs-dev` namespace only

The Argo CD sync policy for our applications is set to *automated* + *prune*, so that child apps are automatically created, synced, and deleted when the manifest is changed. You can change or disable this if you need to. We also configure a webhook against the CI/CD Git repository so that any changes trigger Argo CD to sync all applications; this avoids having to wait for the three-minute sync cycle when CI/CD code changes.

The Git revision can be set to a specific Git commit **Secure Hash Algorithm** (**SHA**) for each child application. A Git SHA is a unique 40-character code computed for every commit to the repository, and is therefore not movable, unlike a tag. This ensures that even if the child app's repository changes, the app will only change when the parent app changes that revision. Alternatively, you can set it to HEAD/master/main or a branch name to keep in sync with that particular branch. It's a good idea to use Git commit SHAs to manage your application versions the closer you are to production environments. Pinning to an exact version for production ensures easier traceability when things go wrong. The structure here is flexible to suit your product team's needs.

Build It – CI/CD for PetBattle

Let's get our hands dirty now by getting down into the weeds with some more techie stuff. How do we go from code to running in production in a repeatable and safe way? If you remember all the way back in *Section 2*, *Establishing the Foundation*, we learned about Derek the DevOps dinosaur and the obstacles we put him through to test his fearsomeness. We will now do the same thing with our PetBattle apps, beginning with the frontend.

The Big Picture

Before we touch a line of code, we always like to keep an eye on the Big Picture. This helps us frame what the tool is and why we are using it, and to scaffold out our pipeline from a very high level. Let's build the details of how we manage the PetBattle source code in this same way. As a subtle reminder, the PetBattle frontend is an Angular application. It was built using Node.js v12 and is deployed to a Red Hat Nginx image.

Continuing our Big Picture from the foundation, let's add the steps we will consider implementing for the PetBattle frontend so we can get it ready for a high frequency of change being pushed through it.

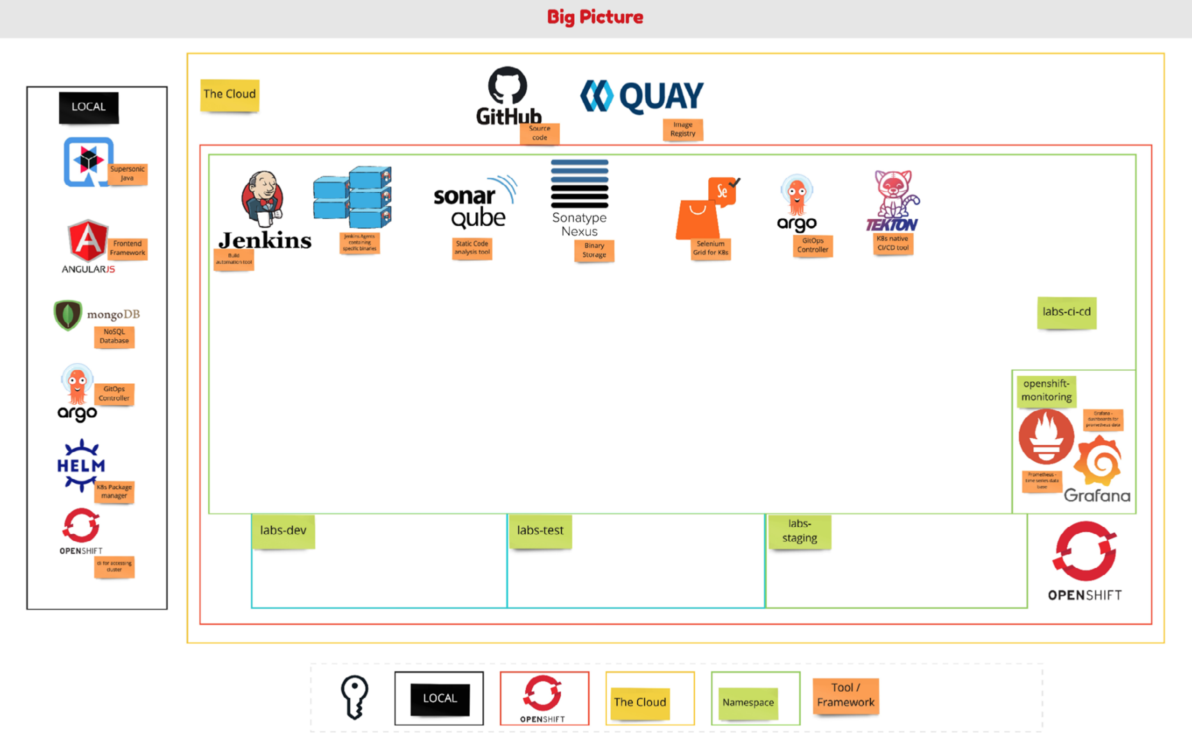


Figure 14.13: The Big Picture including the tools the team thinks they will use

As a quick reminder, our Big Picture from *Section 2*, *Establishing the Foundation*, identified all the tools we might use as depicted in *Figure 14.13*, which include:

- Jenkins: To automate the building and testing of our software
- Nexus: To host our binaries and Helm charts
- Argo CD: To manage our deployments
- SonarQube: To assess our code quality
- Zalenium: For automated browser testing

Now that the tools are in place, let's think about the stages our code should move through for being deployed. A team should start small—what is the minimum amount of automation we need to get code compiling and deployed? It's very important for teams to start small with a basic end-to-end flow of their code; otherwise, things become messy quite quickly, leading to unnecessary complexity and potentially not delivering anything. It's also important because the feedback loop we are creating needs to be fast. We don't want a brilliant, complex process that thinks of everything but takes hours to run! It's not the kind of feedback loop we're trying to create.

We always use three simple stages: Build > Bake > Deploy. A good pattern for engineers to take is to keep to an abstract definition of their pipeline so they can get greater reuse of the pattern across any of their apps, irrespective of the technology they use. Each stage should have a well-defined interface with input and output. Reusing a pipeline definition in this way can lower the context switch when moving between backend and frontend. With this in mind, we can define the stages of our build in the following manner.

The Build

Input: The code base

Output: A "compiled" and unit-tested software artifact

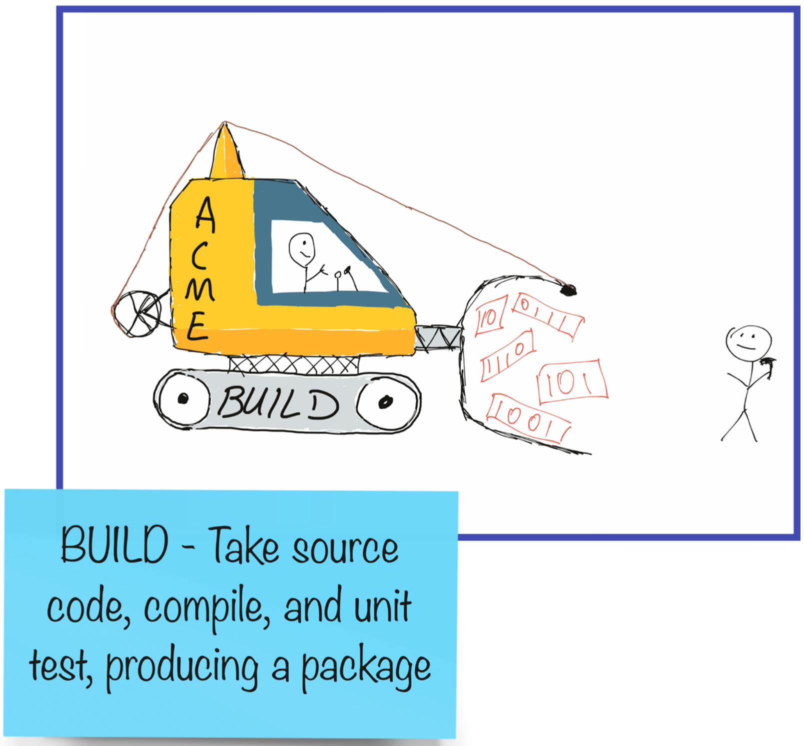


Figure 14.14: The BUILD component from the Big Picture

The build should always take our source code, compile it, and run some unit tests before producing some kind of artifact that will be stored in Nexus. By defining the interface of the build process as lightly as this, it means we can substitute the implementation of what that looks like in each technology or application type. So, for example, when building the PetBattle frontend Angular app, we will use the **Node Package Manager** (**npm**) to complete these steps within the Build stage, but a Java application would likely use Gradle or Maven to achieve the same effect. The hardest work will happen in this stage and it is usually what has the highest dependency on the framework or language that's being used. We will see this in later stages; the technology originally being used becomes less important, so higher reuse of code can occur.

The Bake

Input: A "compiled" software artifact

Output: A tagged container image

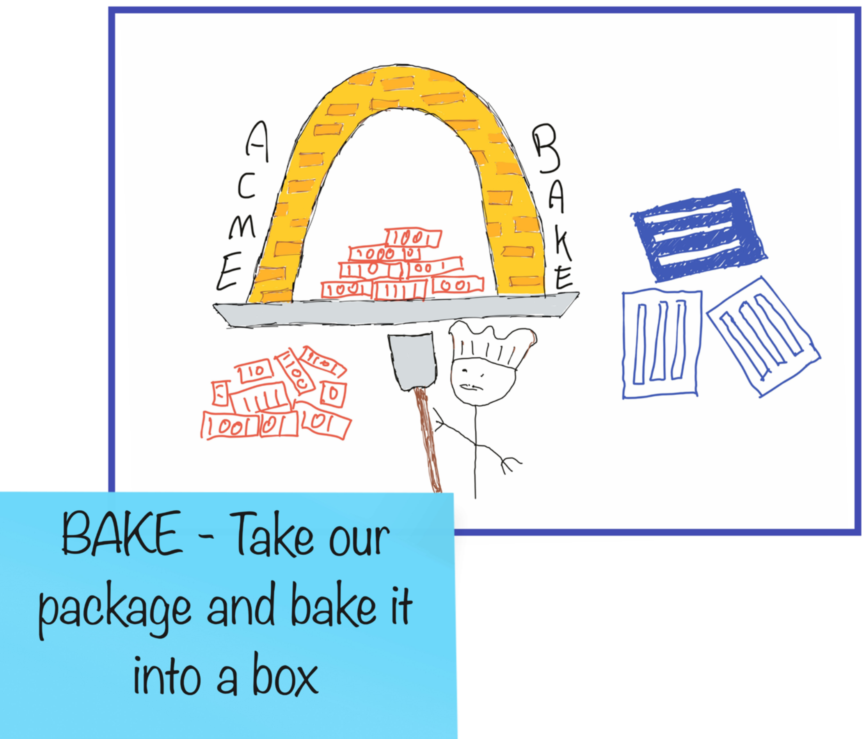


Figure 14.15: The BAKE component from the Big Picture

This is the act of taking our software artifact that was created as an output in the previous step and packaging it into a box, that is, a Linux Container Image. This image is then tagged and stored in a container registry, either one built into OpenShift or an external one. In OpenShift there are many different ways we can achieve this, such as using source-2-image, binary build, or providing a Containerfile/Dockerfile.

The Deploy

Input: A tagged image

Output: A running app in a given environment

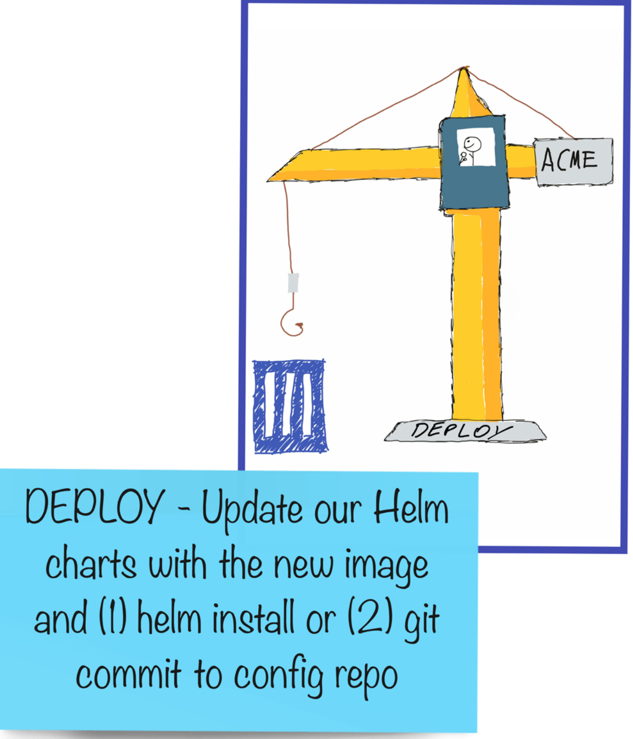


Figure 14.16: The DEPLOY component from the Big Picture

Take the image that has just been pushed to the registry and deploy it along with any other services or configuration required for it to run. Our applications will be packaged as Helm charts, so the deployment will likely have to patch the image referenced in our app's chart. We want our pipeline to support multiple workflows. For feature development, we can just `helm install` into the development namespace. But for release candidates, we should be committing new release information to Git for it to trigger the rollout of changes. The implementation of this workflow is the responsibility of the steps, the lower level of what is being executed. The abstract view of a Deploy should result in a *verified* app deployed on our cluster (and ultimately promoted all the way to production).

The team captures these stages for the applications they're building by adding some nice doodles to their Big Picture. Next, they begin thinking about promoting the application across the environment from test to production. When building applications in containers, we want to ensure the app can run in any environment, so controlling application configuration separately is vital. The team will not want to rebuild the application to target different environments either, so once an image is baked and deployed it needs to be verified before promotion. Let's explore these stages further.

System Test

Input: The app name and version under test

Output: A successful test report and verified app

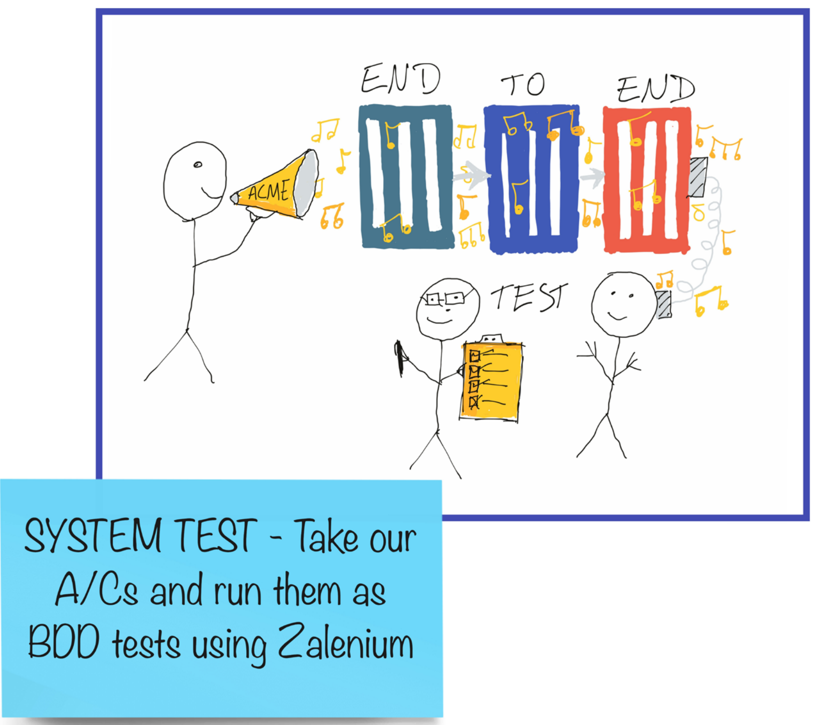


Figure 14.17: The SYSTEM TEST component from the Big Picture

Drive the user behavior within the application via the frontend by verifying whether the app is behaving as expected. If all the connected parts that make up the application are behaving as expected (the microservices, authentication, and frontend) then the app can be signed off and will not need to be rebuilt. Our system test cases for PetBattle will be the acceptance criteria the team has agreed upon. Because of this, we can sign off the application as ready for real-world users. Any component that has changed in the stack should trigger this stage; it is not just the responsibility of the frontend.

Promote

Input: A verified image name and version

Output: Running app in production environment

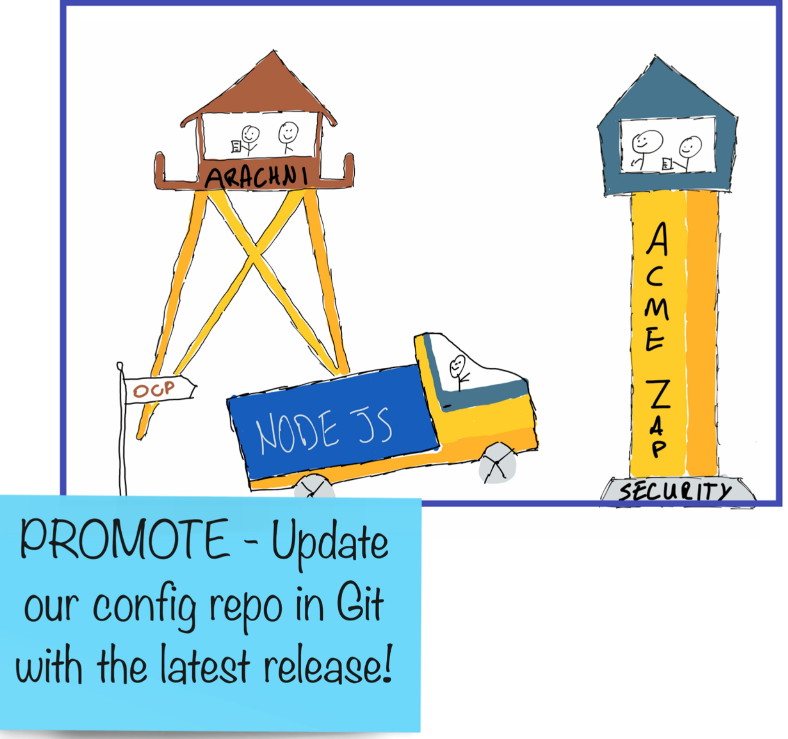


Figure 14.18: The PROMOTE component from the Big Picture

With the application working as expected (based on our passing system test cases), we can now promote the images that make up our app to the new environment, along with their configuration. Of course, in the world of GitOps, this is not a manual rollout of a new deployment but committing the new version and any custom configuration to our configuration repositories, where they will be picked up by Argo CD and deployed.

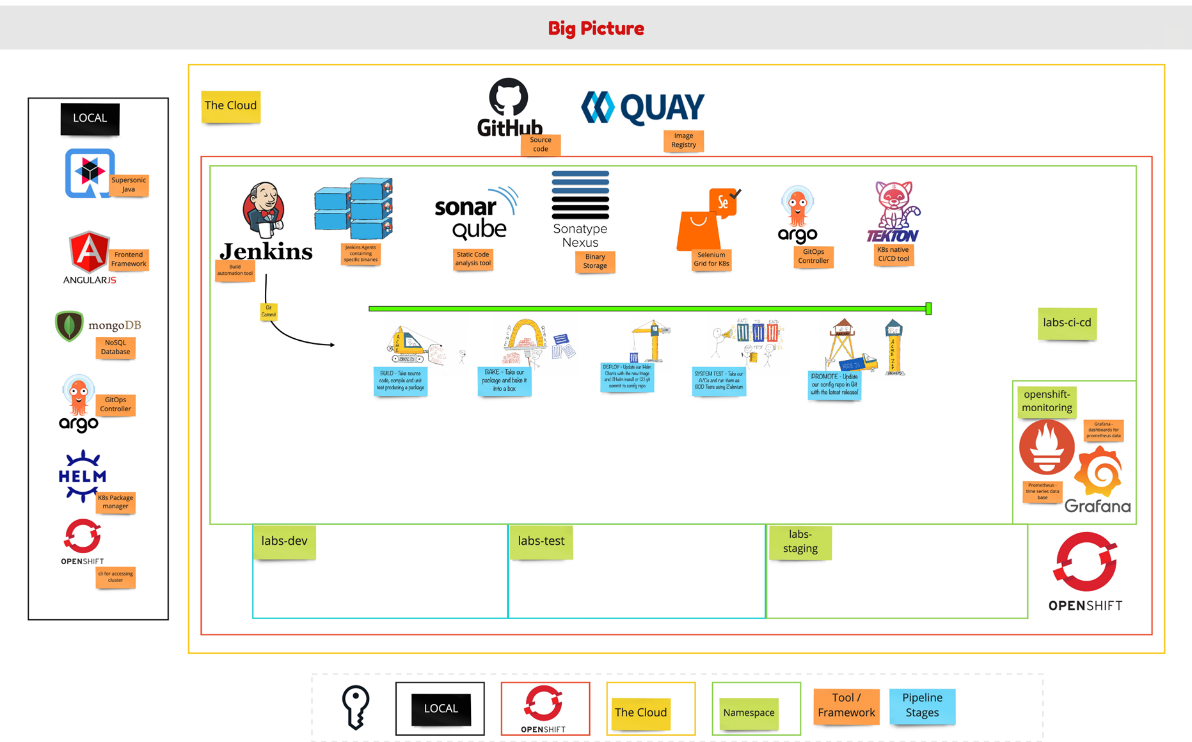


Figure 14.19: The Big Picture including all the stages of the pipeline in place

In *Figure 14.19*, we can see the Big Picture with the stages of the pipeline drawn in. Now that the team knows the stages that their software will pass through on the way across the cluster, they can fill in the lower-level details, the steps. At this stage, the team is looking to see how they can build common pipeline steps, irrespective of the technology they're using. This will provide greater reuse across their software stack but, more importantly, reduce the cognitive load for engineers writing software in multiple technologies. For this, it's a good idea to put on the Big Picture the technology being used. In PetBattle's case, it is Angular and Quarkus (Node.js and Maven for the build tools). They use a new color sticky to write the steps that each service will go through in order to fulfill the interface defined at each stage.

In *Figure 14.20*, we detail what these steps could look like for the Build stage of our pipeline. First, we install the application dependencies. Following this, we test, lint, and compile the code. Finally, we store the successful artifacts in the Nexus repository to use in the next stage, the Bake.

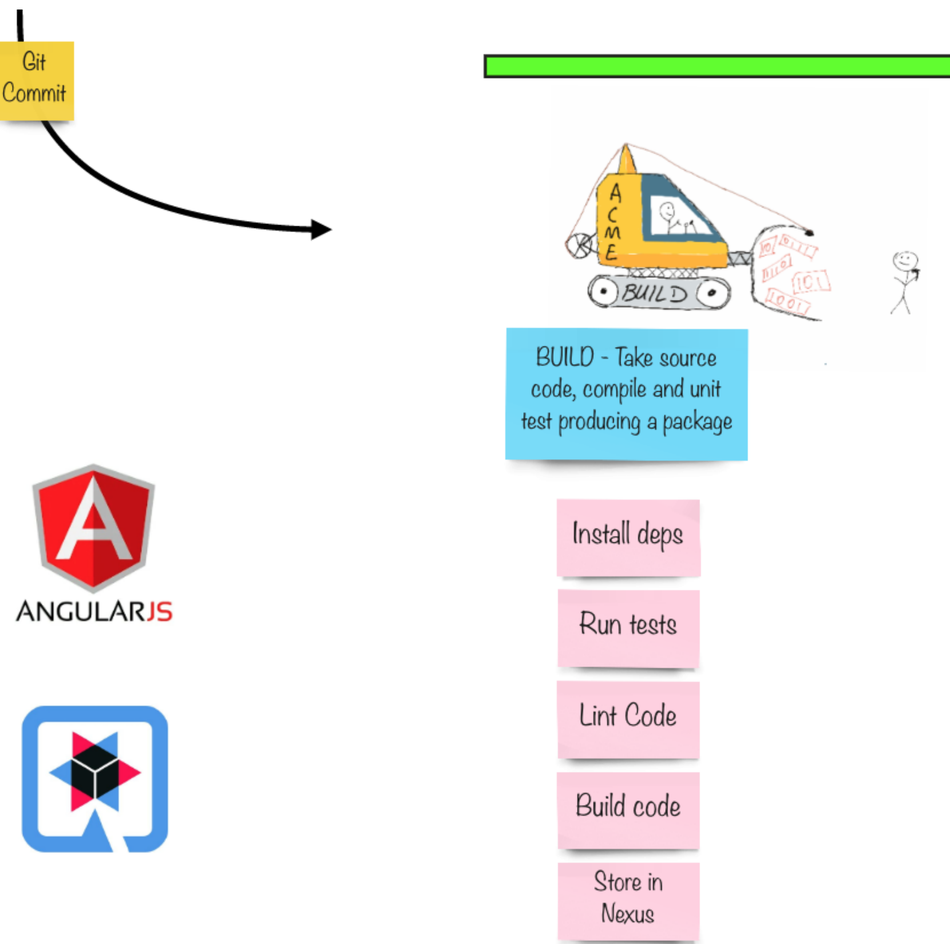


Figure 14.20: The Build stage and the breakdown of its steps

The team continues to flesh out the steps across all the stages. Finally, they add some example containers deployed to each namespace at each stage to give a view of all the components deployed for the PetBattle system to work. This is detailed in *Figure 14.21*:

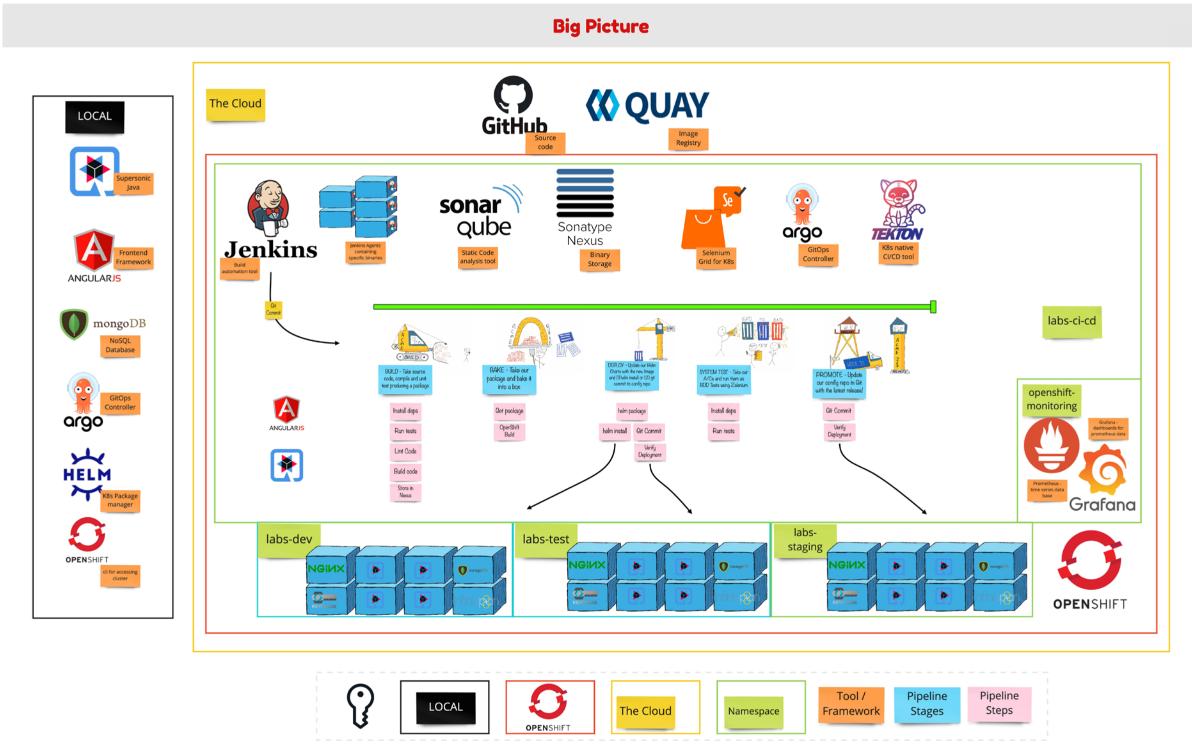


Figure 14.21: The complete Big Picture for our software delivery process

The Big Picture is a helpful practice for getting team alignment on what's in our toolchain and how we use it. It can be a great thing to play back to non-technical team members too, giving them an idea of the complexity and usefulness of being able to repeatedly build and test code. As with all our practices, it's also never done; when a new tool enters our toolchain or we add a new stage in our pipeline, we add it to the Big Picture first. It is the living and breathing documentation of our software delivery process. With the Big Picture complete for now, let's move on to implementing the components it describes.

Choose Your Own Adventure

We understand that there are many ways to do anything and in software development, there are usually hundreds or even more. With this in mind, we hope to meet you, dear reader, where you are now. By this, we mean that the next section will identify two ways of doing the same thing, so take the approach that better fits your own context.

Jenkins is the build tool of choice for lots of companies and developers alike. It has been around for some time and it has its set of quirks for sure. It was never intended to be deployed as a container when it was first conceived. In order to keep things current and have an eye on the future, we've decided to write the code for the Big Picture using both Tekton and Jenkins. Both can easily be tweaked for both frontend and backend development, but for the purposes of this book we will use Jenkins to automate the parts of the Big Picture for our Angular application. For the API, written in Java (Quarkus), we will use Tekton, and go through setting up the same things in a more Kubernetes native way. Both paths are available for the avid reader to play with and get working, but we'll split the narrative this way for illustrative purposes.

So, like you would in a *choose your own adventure* book, you can now pick the path that you would like to follow next. If you're not interested in Jenkins automation, then skip ahead to the Tekton section directly. The code for both options is available in the Git repositories for the book.

Before attempting the pieces in this chapter, make sure to have completed the bootstrap steps in *Chapter 7*, *Open Technical Practices – The Midpoint*, under the *Implementing GitOps – Let's Build the Big Picture With Some Real Working Code!* section. These steps deploy the CI/CD tooling into your cluster using GitOps. The main tools we are going to use in the next sections include Jenkins, Argo CD, and Tekton.

Jenkins–The Frontend

Jenkins is our trusty friend who will do the hard crunching of code—compiling, testing, and so on—on our behalf. In order to get the best out of all the tools in our kit bag, there are a few items we need to configure first. This includes, among other things, managing secrets and adding webhooks to trigger our Jenkins automation as soon as a developer commits their code.

Connect Argo CD to Git

Let's talk about GitOps. We want our Git repositories to be the single source of truth and the Argo CD controller to analyze the differences between what is currently deployed to our cluster and what is stored in our Git repositories. Argo CD can do things based on the difference it sees between the desired state (in Git) and the actual state (in the cluster) such as automatically synchronizing them or sending a notification to say that these two states are not as expected. For example, in Git we may have set version 123 of our application but the cluster currently has version 122 deployed.

To create this connectivity between our configuration repository and Argo CD, we need to create an Argo CD app-of-apps to point to the repository. The app-of-apps pattern is a neat way to describe all elements of a system. Imagine we have an app, named `App-1`, which is our full system. This `App-1` is made up of independently deployable services such as `App-1a`, `App-1b`, `App-1c`, and so on. For PetBattle, we have the whole system that is all of our frontend, APIs, and other services. We also have one of these for our staging and test environments; this allows us to think of our app-of-apps as a suite of applications.

If we clone the `ubiquitous-journey`[7] project that we set up in *Chapter* 7, *Open Technical Practices – The Midpoint*, to bootstrap our cluster, there is another set of charts in here for our application stacks located in *applications/deployments*. When applied, these definitions will create our Argo CD application Custom Resource pointing to our Helm charts that will be created by the running builds on either Tekton or Jenkins.

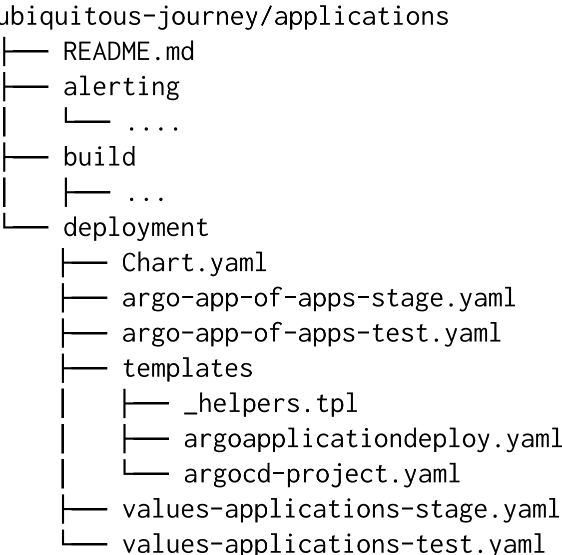
The values files (`values-applications-stage.yaml`) contain the Helm chart version and application version that will be updated by Jenkins on successful builds. We want Argo CD to monitor these values when applying changes to the cluster. These values files also contain our overrides to the base Helm chart for specific environments, for example, the config map that the frontend is configured with to communicate with the services it requires to work properly (`tournament-svc`, `cats-svc`, and so on). The following snippet shows the definition of this. These values will differ between development, testing, and staging, so this pattern gives us the ability to version control the configuration we want the application to use on startup.

```
pet_battle_stage:
    name: pet-battle-stage
    enabled: true
    source: *helm_repo
    chart_name: pet-battle
    sync_policy_automated: true
    destination: labs-staging
    source_ref: 1.0.6
    values:
```

7 https://github.com/petbattle/ubiquitous-journey

```
      fullnameOverride: pet-battle
      image_repository: quay.io
      image_name: pet-battle
      image_namespace: petbattle
      config_map: '{ "catsUrl": "https://pet-battle-api-labs-staging.apps.
hivec.sandbox1405.opentlc.com", "tournamentsUrl": "https://pet-battle-
tournament-labs-staging.apps.hivec.sandbox1405.opentlc.com", "matomoUrl":
"https://matomo-labs-ci-cd.apps.hivec.sandbox1405.opentlc.com/", "keycloak":
{ "url": "https://keycloak-labs-staging.apps.hivec.sandbox1405.opentlc.
com/auth/", "realm": "pbrealm", "clientId": "pbclient", "redirectUri":
"https://pet-battle-labs-staging.apps.hivec.sandbox1405.opentlc.com/*",
"enableLogging": true } }'
      image_version: "master"
    project:
      name: pet-battle-stage
      enabled: true
```

So, when we deploy an Argo CD application pointing to this Git repository, it will find additional apps and so create our app-of-apps pattern. The structure of the repository is trimmed, but you can see that the chart is very basic, having just two templates for creating a project in Argo CD and the application definitions to put inside the project.



```
ubiquitous-journey/applications
├── README.md
├── alerting
│   └── ....
├── build
│   ├── ...
└── deployment
    ├── Chart.yaml
    ├── argo-app-of-apps-stage.yaml
    ├── argo-app-of-apps-test.yaml
    ├── templates
    │   ├── _helpers.tpl
    │   ├── argoapplicationdeploy.yaml
    │   └── argocd-project.yaml
    ├── values-applications-stage.yaml
    └── values-applications-test.yaml
```

We could go to the Argo CD UI and connect it to this repository manually, or use the Argo CD CLI to create the Argo CD application Custom Resource, but let's just run this handy one-liner to connect things up for both our staging and test app-of-apps:

```
# from the root of ubiquitous-journey
$ cd applications/deployment

# install an app-of-apps for each test and staging
$ helm upgrade --install pet-battle-suite-stage -f \
argo-app-of-apps-stage.yaml \
--namespace labs-ci-cd .

$ helm upgrade --install pet-battle-suite-test -f \
argo-app-of-apps-test.yaml \
--namespace labs-ci-cd .
```

With these in place, we should see Argo CD create the app-of-apps definitions in the UI, but it will be unable to sync with the child applications. This is because we have not built them yet! Once they are available, Argo CD will kick in and sync them up for us.

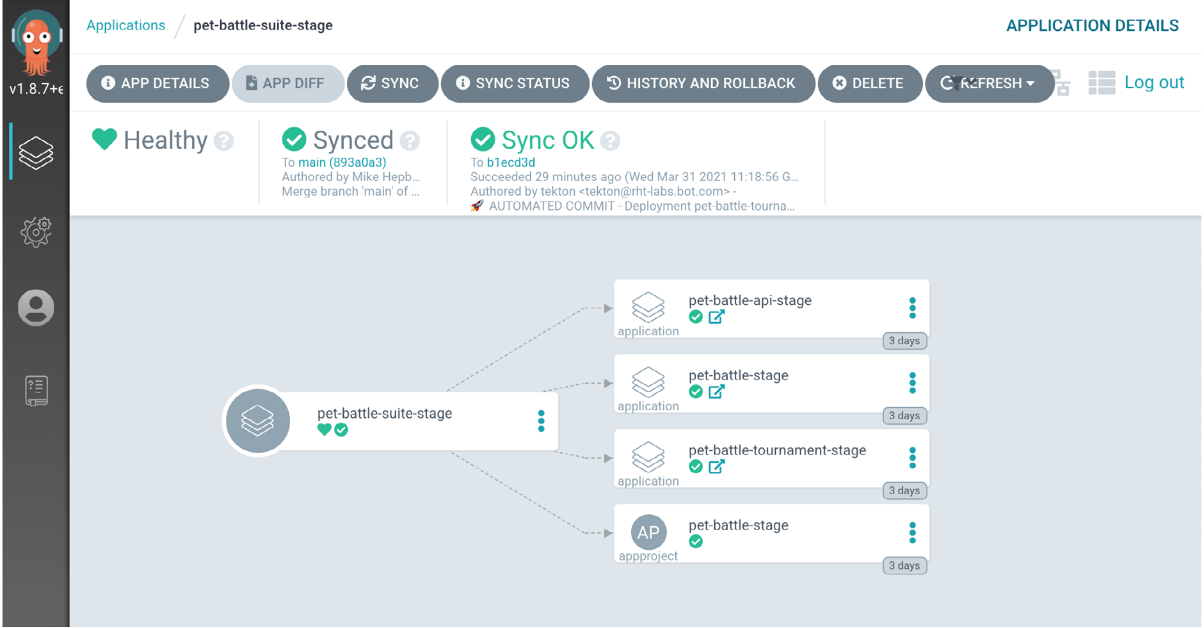


Figure 14.22: Argo CD sync of the pet-battle application suite for the staging environment

To extend this app-of-apps pattern now is very simple. We only need to connect Git to Argo CD this one time. If, after the next few Sprints, the PetBattle team realizes they need to add a new component or service, they can simply extend the values YAML, that is, `values-applications-stage.yaml` or `values-applications-test.yaml` for their staging or test environment, with a reference to the new component chart location and version. For example, for `cool-new-svc`:

```
cool_new_svc_stage:
    name: cool-new-svc-stage
    enabled: true
    source: *helm_repo
    chart_name: cool-new-svc
    sync_policy_automated: true
    destination: labs-staging
    source_ref: 1.0.1 # version of the helm chart
    values:
      fullnameOverride: cool-new-svc
      image_repository: quay.io
      image_name: pet-battle
      image_namespace: petbattle
      image_version: "2.1.3" # version of the application image
    project:
      name: pet-battle-stage
      enabled: true
```

Secrets in Our Pipeline

Jenkins is going to be responsible for compiling our code, pushing the image to a registry, and writing values to Git. This means Jenkins is going to need some secrets! In our case, we're using Quay.io for hosting our images so Jenkins will need the access to be able to push our packaged Container Images to this repository, which requires authentication. If you're following along with forks of the PetBattle repositories and want to create your own running `pet-battle` instance, go ahead and sign up for a free account on https://quay.io/. You can log in with GitHub, Google, or your Red Hat account.

Quay.io

When on Quay, create three new repositories, one for each of the application components that we will be building. You can mark them as public, as private repositories cost money.

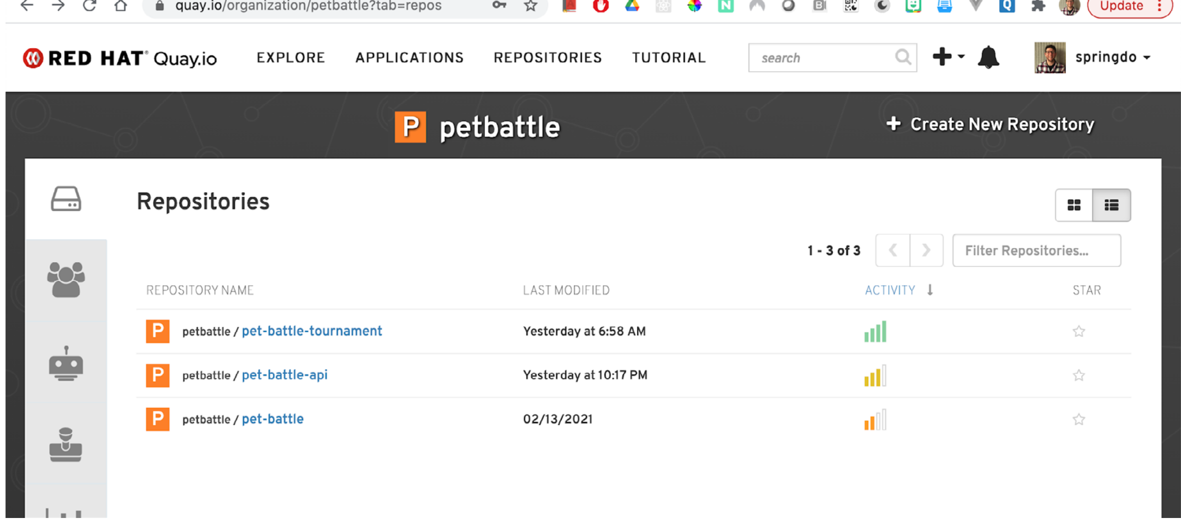


Figure 14.23: PetBattle images in Quay.io

These repositories serve as empty image stores for us to push our images to from within the pipeline. But we need to provide Jenkins with the correct access to be able to push them, so go ahead and hit the robot icon on the UI to create a new service account that Jenkins can use. Give it a sensible name and description for readability.

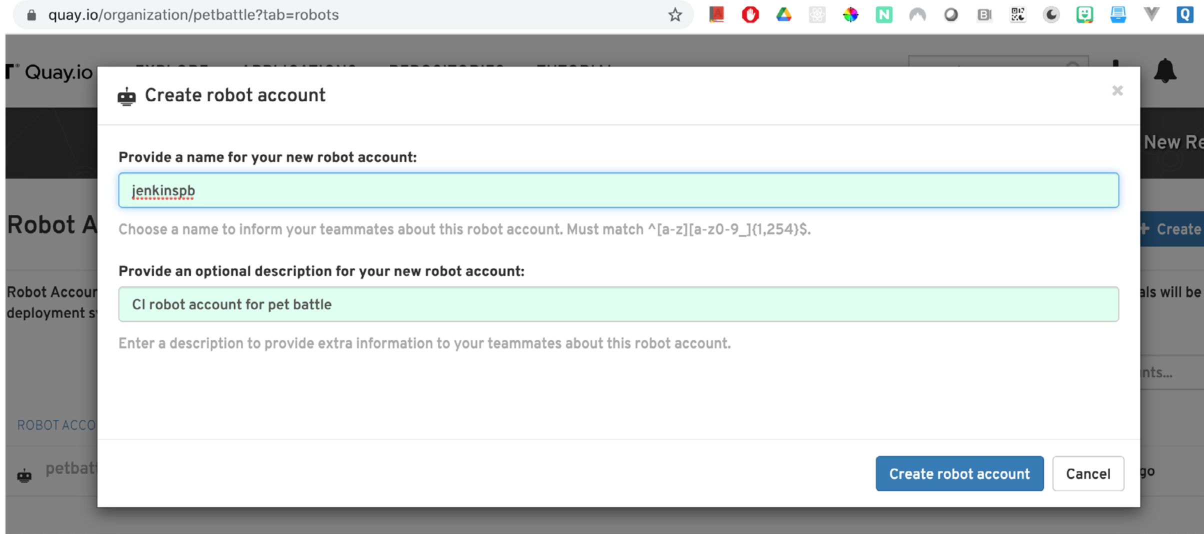


Figure 14.24: Robots in Quay.io

We are going to mark all the repositories we created previously as **Write** by this robot. Hit **Add permissions**:

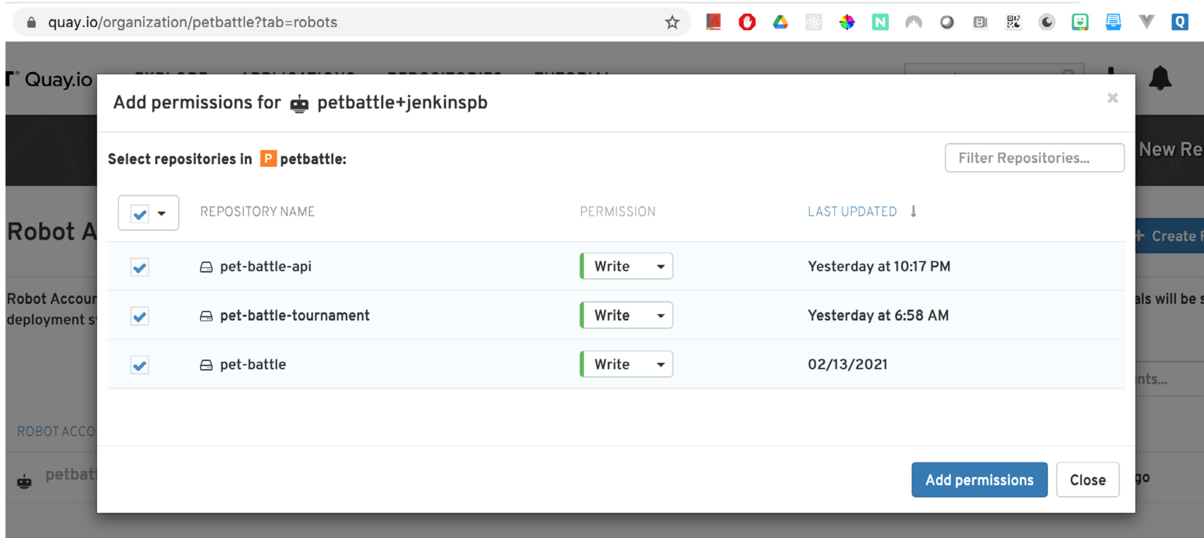


Figure 14.25: Robot RBAC in Quay.io

Now that the repositories and robot account have been created, we can download the secret to be used in our pipelines! Hit the cog on the side of the secret name and select **View Credentials**.

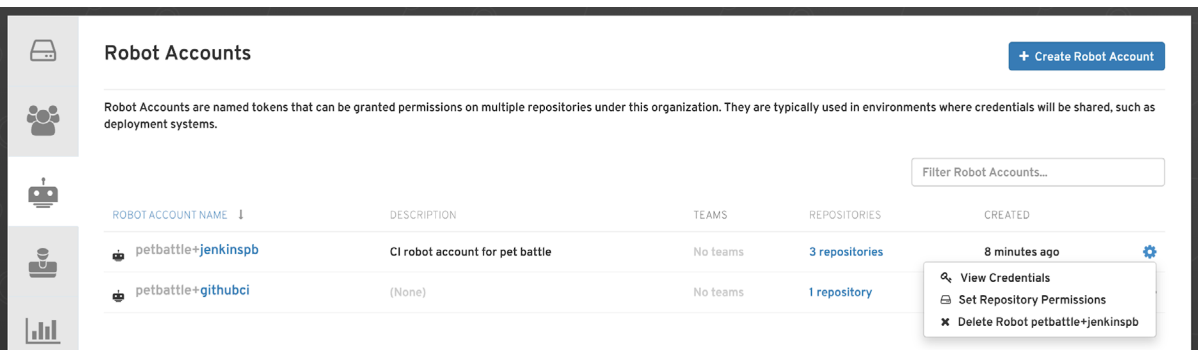


Figure 14.26: How to view the robot credentials in Quay.io

On the page that pops up, download the Kubernetes YAML and store it in your fork of `pet-battle`.

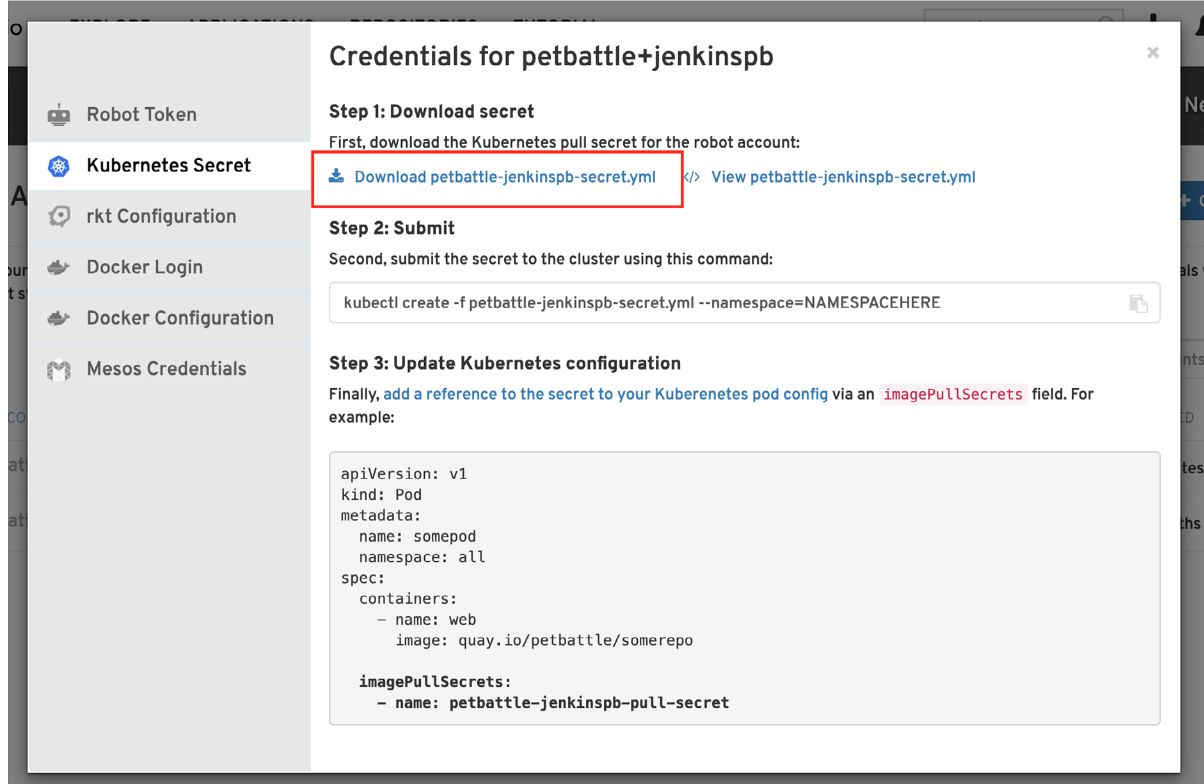


Figure 14.27: Downloading the Kubernetes secret

We can apply it to our cluster (ensure you are logged in first):

```
$ oc apply -n labs-ci-cd -f petbattle-jenkinspb-secret.yml
```

GitHub

A secret is also required for Jenkins to be able to push updates to our Helm values files stored in Git. The values files for our applications will contain the properties we want to pass to our templates, such as ConfigMap variables, or locations of images, such as Quay.io. Our values files for the deployment of our applications will also hold a reference to the image version (that is, the SemVer of our app, such as 1.0.1) to be deployed by patching our DeploymentConfigs. We don't want to manually update this but have a robot (Jenkins) update this when there has been a successful build. Therefore, this secret will be needed to write these changes in versions to our configured repositories, which are being pointed out by Argo CD. We track version changes across all our environments in this way because, after all, if it's not in Git, it's not real.

To create a secret for GitHub, simply go to the Developer Settings view. While logged into GitHub, that's **Settings > Developer Settings > Personal access tokens** or https://github.com/settings/tokens for the lazy. Create a new **Personal Access Token** (**PAT**); this can be used to authenticate and push code to the repository. Give it a sensible name and allow it to have repository access.

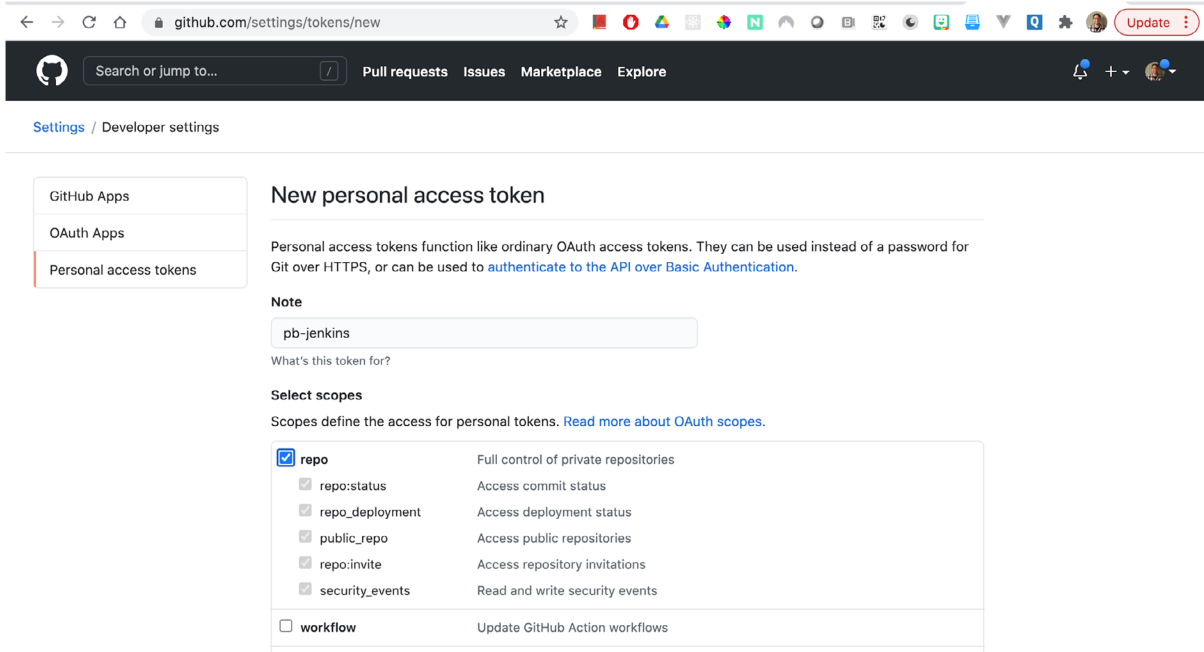


Figure 14.28: GitHub personal access token permissions

Save the token's value, as you won't be able to access it again without generating a new one. With the token in place, we can create a secret in Jenkins by adding it to a `basic-auth` secret. In order for Jenkins, which is running in the same namespace as where this secret will be created, to be able to consume the value of the secret, we can apply a special annotation, `credential.sync.jenkins.openshift.io: "true"`. This little piece of magic will allow any credentials to be updated in Jenkins by only updating the secret!

Update the secret with your values for `GITHUB_TOKEN` and `GITHUB_USERNAME` if you're following along in your own fork and apply them to the cluster:

```
$ cat <<EOF | oc apply -f-
apiVersion: v1
stringData:
  password: GITHUB_TOKEN
  username: GITHUB_USERNAME
kind: Secret
metadata:
  labels:
    credential.sync.jenkins.openshift.io: "true"
  name: git-auth
  namespace: labs-ci-cd
type: kubernetes.io/basic-auth
EOF
```

SealedSecrets

You might be thinking that these secrets should probably be stored somewhere safe—and you're right! If you want to explore the idea of storing the secrets in Git so they too are *GitOpsy* (yes, I did just invent a word there), then we could use SealedSecrets by Bitnami. It provides a controller for encrypting secrets, allowing us to store them as plain text. This means we can commit them to Git! Through the magic of the SealedSecret Custom Resource, it decrypts the SealedSecret, and creates a regular Kubernetes secret on your behalf. We've written our Jenkins Helm chart to accept SealedSecrets for this very reason!

You can deploy SealedSecrets to the cluster by enabling it in the Ubiquitous Journey Git project. Open up `bootstrap/values-bootstrap.yaml`. It's as simple as changing the `enabled` flag to `true` and, of course, Git committing the changes. This will resync with Argo CD and create an instance of Bitnami SealedSecrets in your cluster, by default in the `labs-ci-cd` namespace. Because this is a new component we're adding to our tooling, we should of course also update our Big Picture with the tool and a sentence to describe how we use it.

```
sealed-secrets:
  # Disabled by default
  enabled: true
  nameOverride: sealed-secrets
```

Once the controller has been created, we can seal our secrets by following these few steps:

1. Install kubeseal using the instructions found on their GitHub releases page: https://github.com/bitnami-labs/sealed-secrets/releases.
2. Log in to the cluster where SealedSecrets is deployed and take note of the namespace (in our case this defaults to labs-ci-cd).
3. Process your existing secret using the kubeseal command-line utility. It is important to set the correct namespace otherwise the secret will not be unsealed. In this case, we're going to seal it as super-dooper-secret. It should look something like this:

```
# create secret file from step 3
$ cat << EOF > /tmp/super-dooper.yaml
---
apiVersion: v1
kind: Secret
metadata:
  name: super-dooper
  labels:
    credential.sync.jenkins.openshift.io: "true"
type: "kubernetes.io/basic-auth"
stringData:
  password: "myGitHubToken"
  username:  "donal"
EOF

# encrypt the secret
$ kubeseal < /tmp/super-dooper.yaml > /tmp/sealed-super-dooper.yaml \
  -n labs-ci-cd \
  --controller-namespace labs-ci-cd \
  --controller-name sealed-secrets \
  -o yaml
```

4. You can now apply that secret straight to the cluster for validation, but you *should* add it to the cluster using Argo CD by committing it to Git. If it's not in Git, it's not real. Here, we can see what the SealedSecret looks like before it's applied to the cluster. As you can see, it's a very large, encrypted string for each variable we sealed:

```
# have a look at the sealed secret
$ cat /tmp/sealed-super-dooper.yaml

apiVersion: bitnami.com/v1alpha1
kind: SealedSecret
metadata:
  creationTimestamp: null
  name: super-dooper
  namespace: labs-ci-cd
spec:
  encryptedData:
    password: AgC6NyZa2to2MtKbXYxJBCOfxmnSQ4PJgV8KGdDRawWstj24FIEm5YCyH6n/
BXq9DEPIJL4IshLb2+/kONKHMhKy0CW5iGVadi13GcvO7lxZpVLeVr4T3nc/AqDwPrZ2KdzAI-
62h/7O4o2htRWrYnKqzmUqdESzMxWCK9dl7HZyArGadvwrH98iR48avsaNWJRvhMEDD6EM-
jS5yQ2vJYFMcMzOVyMtbD4f8e3jK7OO+vqoXsHtiuHk4HB63BZZqreiDcFgZMGaD6Bo6FyMSs/
tbkBjttiRvP5zZJ5fqC8IEgbZeuwhJ1eVOeKs/2xGBUMoEiYo6cKaUOqV9k130K2wcdX-
gN8B25phkRK9DpO23LoF/7/uLwNn01pCcxAxm1/2kvX24uPLtirmg1rQ03E9qrnlvyky-
J+9G3QBNtIlsiuoYmEYogZCSRZX29Cm0GWLolYPhlhMDDN6VQI6ktKCH6ubMcbh888Gn2K-
F8NzpQvV5wN9mQVFMR8+wNVkLGsaN+EEdgAc2CmiajIXur3zu4Menq3iWzJcWHdyT-
NlROpJeFH9qyfJLzbkWinPyzyBZEXeiZVKZ/ZAYEvXpyHAUngbnNnUO8HBwsLHb//
uYEzWRufIJezCy9PYxUVSBNIdfPybuCSeb87Bgry/+5D5aUjrqLuKJUhsLWIL3waHyvQswU-
jCQlcgFA7OZ9lwMqkDUYy9SnYatIZ98kf1Z6DA==
    username: AgDY4NgxKug07A+jZ63h0Rdisfm6o7kVaKaiaPek9ZOiHsox1A0P4k-
lYaK/7cTEyOCpFVC/2nx00TX6F2KbA1GsRHkjnU/79nOkYWqsWWTU32c/0Re8sSEIPX7aVgR/
sMXYeWyRediRogA23xFcFzIFSvw4fZ2XpeX0BZNPbMdwZv2b+j/cjW8Po75B5gqbjwhMy-
H36QUApnjmoWmutLONVgAnHVM2rBr1Kx4wgxyy+hdmj+6ZkgMBckd53lMVX0unRVW93I-
j2eDcxTwN+HvVY7nBDmxVHuYAt6t31+DXpqBew10kNDxd8Xw2MpUFDb3JpMwIVtTnt-
mgeoyCHmo7nCYzQkGhwdrEYzoLVQBq+jf0Wmu3YRpEzZbegdTU3QfS1J7XM+86pAF6g-
cgbmrhpguGkU+PwnzPMxGNkq445oEPpvRemftjyFf7A8C+bZ90lrvVzZsfOue8WdXKm-
66vZoYuMPqA2o2HQV0IraaNGYPt9FmiAuXqWhzKsSVsbURXUUOaZIPAyX1z5V1reRz+gs/
cGHYKbmUua7XOFQr32siANI1IkRPi9cT+9iP9GGdq5RzZL75cJGFV8BorZ3CMADGC+skrFKO-
ExFvSrvofBnODB/xnPuirzsnQPcxtdvIz+sCv4M8qG2j0ASH1DBLLF7vMP9rLBgA1sPtzqX-
0CBakjuOjYDqpbXaKqHrM6kdTuBvO7tTDpAYA==
  template:
    metadata:
      creationTimestamp: null
```

```
    labels:
      credential.sync.jenkins.openshift.io: "true"
    name: super-dooper
    namespace: labs-ci-cd
  type: kubernetes.io/basic-auth

# apply it to the cluster
$ cat /tmp/sealed-super-dooper.yaml | oc apply -f- -n labs-ci-cd

sealedsecret.bitnami.com/super-dooper configured
```

5. To *GitOpsify* (yes, again I did just make that up), open up your Jenkins configuration in `ubiquitous-journey/values-tooling.yaml`. Set your values on Jenkins `sealed_secrets` as follows using the output of the secret generation step to add the encrypted information to each key. The example here is trimmed for readability:

```
- name: jenkins
  enabled: true
  source: https://github.com/redhat-cop/helm-charts.git
  ...
  values:
    ...
    sealed_secrets:
      - name: super-dooper
        password: AgAD+uOI5aCI9YKU2NYt2p7as.....
        username: AgCmeFkNTa0tOvXdI+lEjdJmV5u7FVUcn86SFxiUAF6y.....
```

6. If you've already manually applied the secret in *Step* 4, delete it by running `cat /tmp/sealed-super-dooper.yaml | oc delete -f- -n labs-ci-cd`. Then `Git commit` these changes so they are available to Jenkins and, more importantly, stored in Git. In Argo CD, we should see that the SealedSecret generated a regular secret.

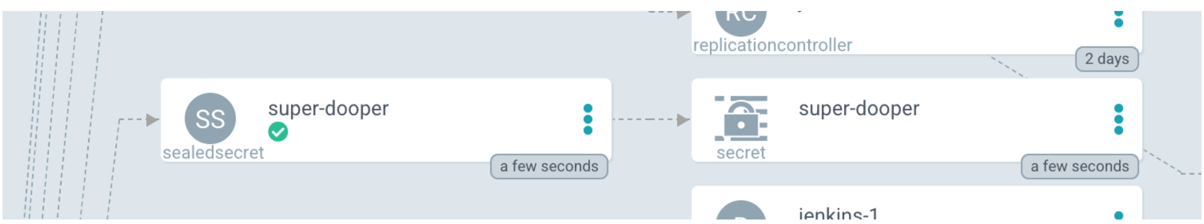


Figure 14.29: SealedSecrets from Argo CD

7. In Jenkins, we should see all that the synchronized secrets using the `magic` annotation (`credential.sync.jenkins.openshift.io: "true"`) have become available.

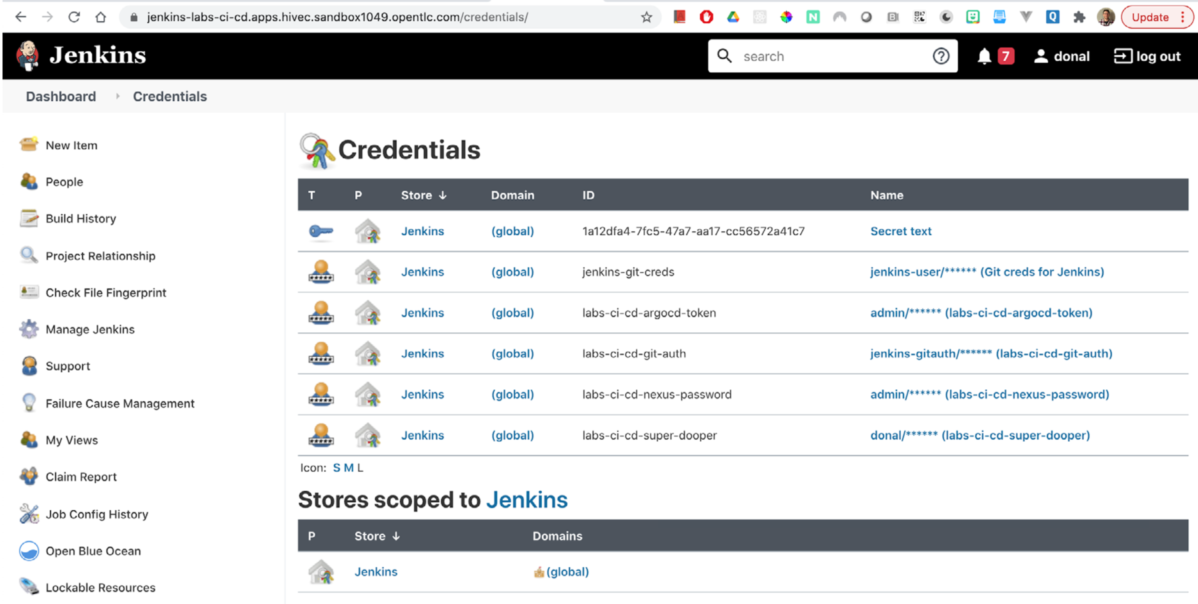


Figure 14.30: Jenkins secrets automatically loaded from Kubernetes

For simplicity here, we will continue without having sealed the secrets; the topic of secrets and GitOps has been included only for illustrative purposes.

The Anatomy of a Jenkinsfile

Some of you may be familiar with a `Jenkinsfile`, but for those who are not, let's take a look at the anatomy of one. A `Jenkinsfile` is just a simple **domain-specific language** (**DSL**) that Jenkins knows how to interpret and build our pipelines from. It's the blueprint for the tasks we want to carry out and the order in which they come. Historically, people manually configured Jenkins by creating jobs in the UI, but as people started to do these things at scale, repeatability and maintenance became an issue. I remember when on my first project, one of the developers' code kept failing the build, so he just edited the job to remove that step. Because we had no authorization or traceability around who could change what on Jenkins, this change was not noticed for some time, until things began to break because of it. Fast-forward to now, and we have the `Jenkinsfile`. It is an *everything-as-code* practice that defines the sequence of things we want our pipeline to execute in order.

The `Jenkinsfile` is made up of a pipeline definition with a collection of blocks, as the following is from our PetBattle frontend. If you're curious as to where this file is, you will find it at the root of the project in Git. *Figure 14.31* is trimmed a little for simplicity:

```
Jenkinsfile
1    pipeline {
2      agent {
3        label "master"
4      }
5  >   environment {…
27     }
28     // The options directive is for configuration that applies to the whole job.
29 >   options {…
34     }
35
36     stages {
37       stage(' Perpare Environment') {
38         failFast true
39         parallel {
40 >         stage(" Release Build") {…
60           }
61 >         stage(" Sandbox Build") {…
86           }
87         }
88       }
89
90       stage(" Build (Compile App)") {
91         agent { label "jenkins-agent-npm" }
92         steps {
93           echo '### Running build ###'
94           sh '''
95               npm run build
96           '''
97         }
98         // Post can be used both on individual stages and for the entire build.
99         post {
100 >        always {…
110          }
111        }
112      }
113
114 >    stage(" Bake (OpenShift Build)") {…
150      }
151 >    stage(" Deploy - Helm Package") {…
182      }        You, 4 days ago • Jenkinsfile for pet battle
183 >    stage(" Deploy - App") {…
241      }
242    }
```

Figure 14.31: Anatomy of a Jenkinsfile

The key aspects of a `Jenkinsfile` DSL are:

- `pipeline {}` is how all declarative Jenkins pipelines begin.
- `environment {}` defines the environment variables to be used across all Build stages. Global variables can be defined here.
- `options {}` contains specific job specs you want to run globally across jobs; for example, setting the terminal color or the default timeout.
- `stages {}` encapsulates the collection of blocks our pipeline will go through, that is, the `stage`.
- `stage {}`: All jobs must have at least one stage. This is the logical part of the build that will be executed, such as `bake-image`, and contains the steps, agents, and other stage-specific configurations.
- `agent {}` specifies the node that the build should be run on, for example, `jenkins-agent-npm`.
- `steps {}`: Each stage has one or more steps involved. These could be executing shell commands, scripts, Git checkout, and so on.
- `post {}` is used to specify the post-build actions. Jenkins' declarative pipeline syntax provides very useful callbacks for `success`, `failure`, and `always`, which are useful for controlling the job flow or processing reports after a command is executed.
- `when {}` is used for flow control. It can be used at the stage level, as well as to stop the pipeline from entering that stage; for example, when branch is master, deploy to test environment.
- `parallel {}` is used to execute some blocks simultaneously. By default, Jenkins executes each stage sequentially. If things can be done in parallel, then they should, as it will accelerate the feedback loop for the development team.

For us, we are creating the components in the Big Picture, which are Build > Bake > Deploy.

The Build should always take the source code, compile it, run some linting (static code checking) and testing before producing a package, and store it in Nexus. We should produce test reports and have them interpreted by Jenkins when deciding to fail the build or not. We are building an Angular application, but Jenkins does not know how to execute `npm` or other JavaScript-based commands, so we need to tell it to use the agent that contains the `npm` binary. This is where the agents that we bootstrapped to the Jenkins deployment will come in handy. Each agent that is built extends the base agent image with the binary we need (that is, `npm`) and is pushed to the cluster. This ImageStream is then labeled `role=jenkins-slave` to make it automatically discoverable by Jenkins if they are running in the same namespace. This means that for us to use this, we just need to configure our Jenkins stage to use `agent { label "jenkins-agent-npm" }`.

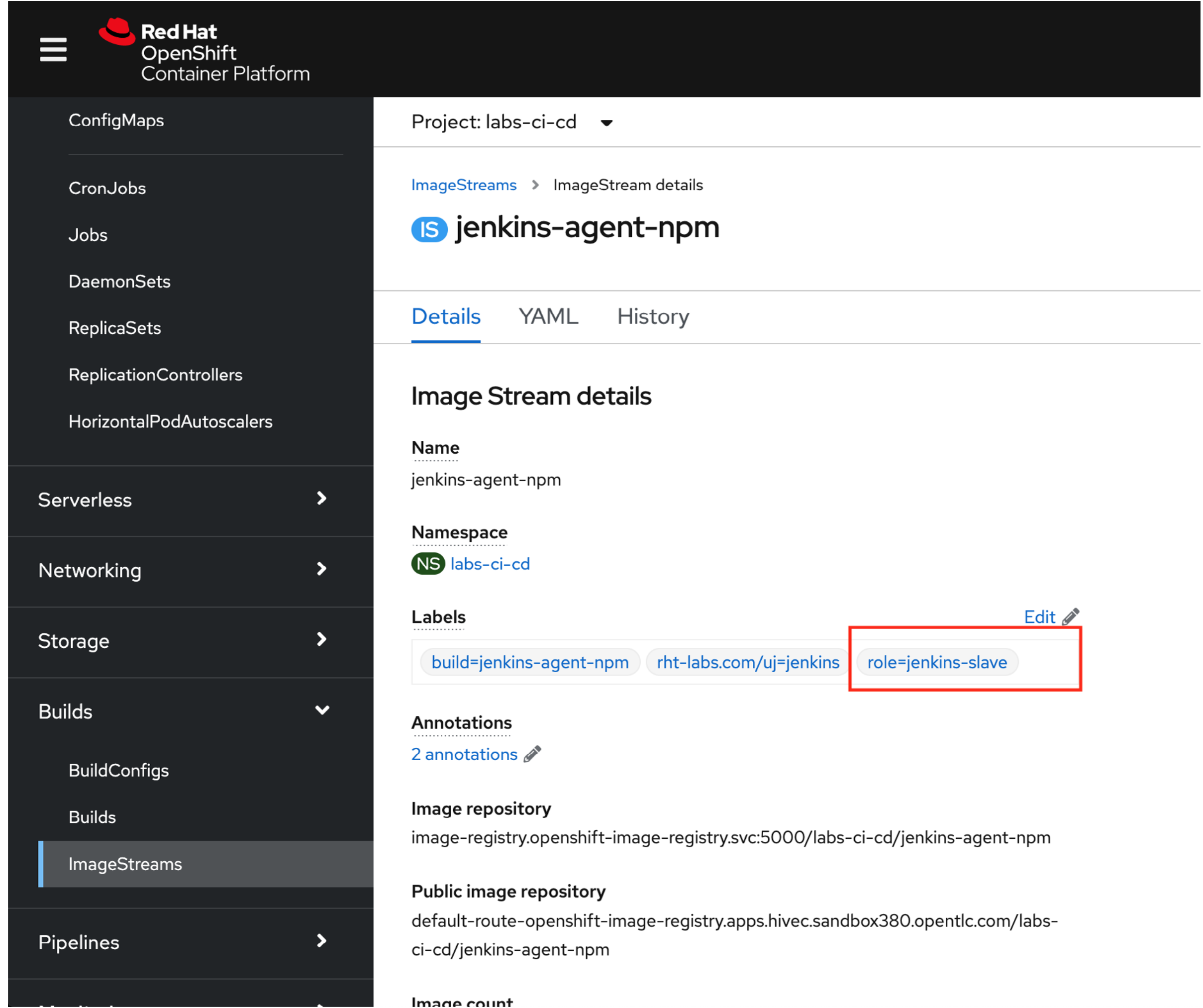


Figure 14.32: Jenkins agent discovery magic

The Build stage will use this agent and execute some steps. The first thing is to capture the app's version to be used throughout the pipeline by reading the app's manifest (`pom.xml` for Java or `package.json` for Node). This version is then used on all generated artifacts, including our image and Helm chart version, and should follow SemVer (for example, `<major>.<minor>.<patch> = 1.0.1`). We will then pull our dependencies, run our tests, lint, and build our code before publishing the results to Jenkins and the package to Nexus.

This will display in the Jenkins declarative pipeline like so:

```
stage("Build (Compile App)") {
  agent { label "jenkins-agent-npm" }
  steps {
    script {
        env.VERSION = sh(returnStdout: true, script: "npm run version
--silent").trim()
        env.PACKAGE = "${APP_NAME}-${VERSION}.tar.gz"
    }
    sh 'printenv'
    echo '### Install deps ###'
    // sh 'npm install'
    sh 'npm ci --registry http://sonatype-nexus-service:${SONATYPE_NEXUS_
SERVICE_SERVICE_PORT}/repository/labs-npm'
    echo '### Running linter ###'
    sh 'npm run lint'
    echo '### Running tests ###'
    sh 'npm run test:ci'
    echo '### Running build ###'
    sh '''
        npm run build
    '''
    echo '### Packaging App for Nexus ###'
    sh '''
        tar -zcvf ${PACKAGE} dist Dockerfile nginx.conf
        curl -v -f -u ${NEXUS_CREDS} --upload-file ${PACKAGE}
http://${SONATYPE_NEXUS_SERVICE_SERVICE_HOST}:${SONATYPE_NEXUS_SERVICE_
SERVICE_PORT}/repository/${NEXUS_REPO_NAME}/${APP_NAME}/${PACKAGE}
    '''
  }
  post {
    always {
      junit 'junit.xml'
      publishHTML target: [
        allowMissing: true,
        alwaysLinkToLastBuild: false,
        keepAll: false,
        reportDir: 'reports/lcov-report',
        reportFiles: 'index.html',
        reportName: 'Code Coverage'
      ]
    }
  }
}
```

Our Bake will always take the output of the previous step, in this case, the package stored in Nexus, and pop it into a container. In our case, we will be running an OpenShift build. This will result in the package being added to the base container and pushed to a repository. If we are executing a sandbox build, say some new feature on a branch, then we are not concerned with pushing the image externally—so we can use the internal registry for OpenShift. If this build is a release candidate then we'll push into Quay.io (our external registry for storing images). The breakdown of the steps for a Bake is found in the Git repository that accompanies this book: https://github.com/petbattle/pet-battle/blob/master/Jenkinsfile.

From a bird's-eye view, the idea is to get the package from Nexus and then create an OpenShift `BuildConfig` with a binary build and pass the package to it. You should then see the build execute in the OpenShift cluster.

```
130
131 >     stage("🧁 Bake (OpenShift Build)") {⋯
167       }
168
169 >     stage("🏗 Deploy - Helm Package") {⋯
200       }
201
202       stage("🏗 Deploy - App") {
203         failFast true
204         parallel {
205 >         stage("🏖 Sandbox - Helm Install"){⋯
221           }
222 >         stage("🧪 TestEnv - ArgoCD Git Commit") {⋯
258           }
259         }
260       }
261     }
262   }       You, 10 months ago • 🦄 Headless e2e and Jenkinsfile merge 🔥
```

Figure 14.33: Jenkins stages outline for Bake and Deploy

The deployment will take the application that has just been packaged up with its dependencies and deploy it to our cluster. Initially, we will push the application to our `labs-test` environment. We want to package the application and its Kubernetes resources as a Helm chart, so for the deployment we will patch the version of the application referenced in the values file with the latest release. For this reason, our Deploy stage is broken down into two parts.

The first one patches the Helm chart with the new image information, as well as any repository configuration, such as where to find the image we just Baked! This is then stored in Nexus, which can be used as a Helm chart repository.

Secondly, it will install this Helm chart. Depending on what branch we're on, this behavior of how the application will be deployed differs. If we're building on `master` or `main`, it is a release candidate, so there is no more oc applying some configuration—this is GitOps land! Instead, we can commit the latest changes to our Argo CD config repository (Ubiquitous Journey). The commits on this repository should be mostly automated if we're doing this the right way. Managing our apps this way makes rollback easy—all we have to do is Git revert!

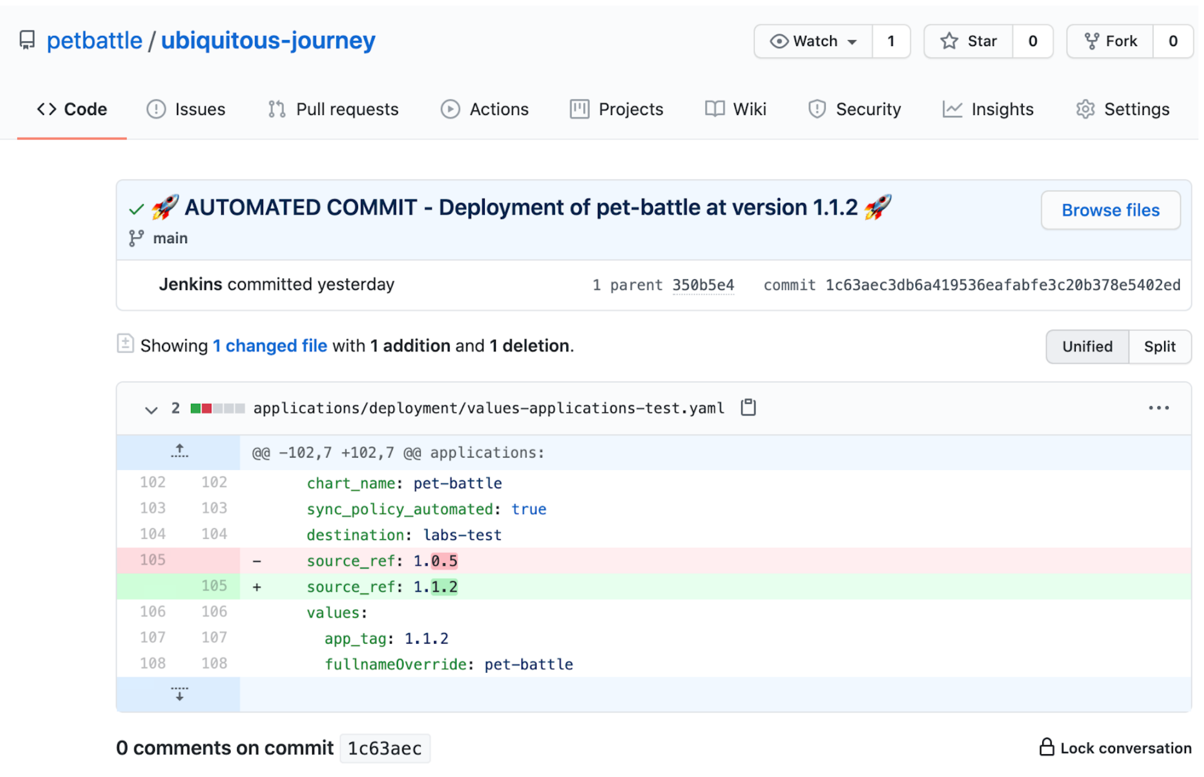


Figure 14.34: Jenkins automated commit of the new version from a pipeline run

Branching

Our pipeline is designed to work on *multibranch*, creating new pipeline instances for every branch that is committed to in Git. It is intended to have slightly different behavior on each branch. In our world, anything that gets merged to `master` or `main` is deemed to be a release candidate. This means that when a developer is ready to merge their code, they would amend the `package.json` version (or `pom.xml` version for Java projects) with the new release they want to try and get all the way through the pipeline to production. We could automate the version management, but because our workflow has always been easier, a developer will do this management, as they are best placed to decide whether it's a patch, a minor, or a major release.

This means that anything not on the `main` or `master` branch is deemed to be a sandbox execution of the pipeline. If something is a sandbox build, it is there to provide fast feedback to the developers of the current state of development in that feature. It can also act as a warning to other engineers that something is not ready to be merged if it's failing. The sandbox builds should be thought of as ephemeral—we're not interested in keeping them hanging around—hence we make some key changes to the pipeline to accommodate this:

1. **Internal registry**: If our built image is pushed to our external repository, it will become clogged up and messy with unnecessary images. Every time a developer commits to any branch it would create new images, so it can introduce a cleanup headache; hence we use the internal registry, which automatically prunes old images for us. We only use the external registry when we know a release could go all the way to production.

2. **Helm install**: For our deployments, we're not interested in bringing in a heavyweight tool like Argo CD to manage the development/sandbox deployments. It's unnecessary, so we just use Jenkins to execute a Helm install instead. This will verify that our app can deploy as we expect. We use Argo CD and GitOps to manage the deployments in test and staging environments, but any lower environments we should also treat as ephemeral (as we should test and staging too).

This approach allows us to support many different types of Git workflow. We can support GitHub Flow, Gitflow, and Trunk, all via the same consistent approach to the pipelines.

Webhooks

Before we actually trigger Jenkins to build things for us, it's important to add a few webhooks to make our development faster. We need two, one for the Argo CD config repo and one for Jenkins in our source code repository.

When we commit a new change to the Git repositories that Argo CD is watching for, it polls. The poll time is configurable, but who can be bothered to wait at all? Argo CD allows you to configure a webhook to tell it to initiate a sync when a change has been made.

This is particularly important if we want things to happen after Argo CD has worked its magic, such as in a system test. Our pipeline in Jenkins runs synchronously, but Argo CD is asynchronous and therefore anything we can do to reduce the wait between these behaviors is critical.

On GitHub, we can configure the webhook for Ubiquitous Journey to trigger Argo CD whenever the repository updates. On GitHub, add the webhook with the address of our Argo CD server followed by `/api/webhook`.

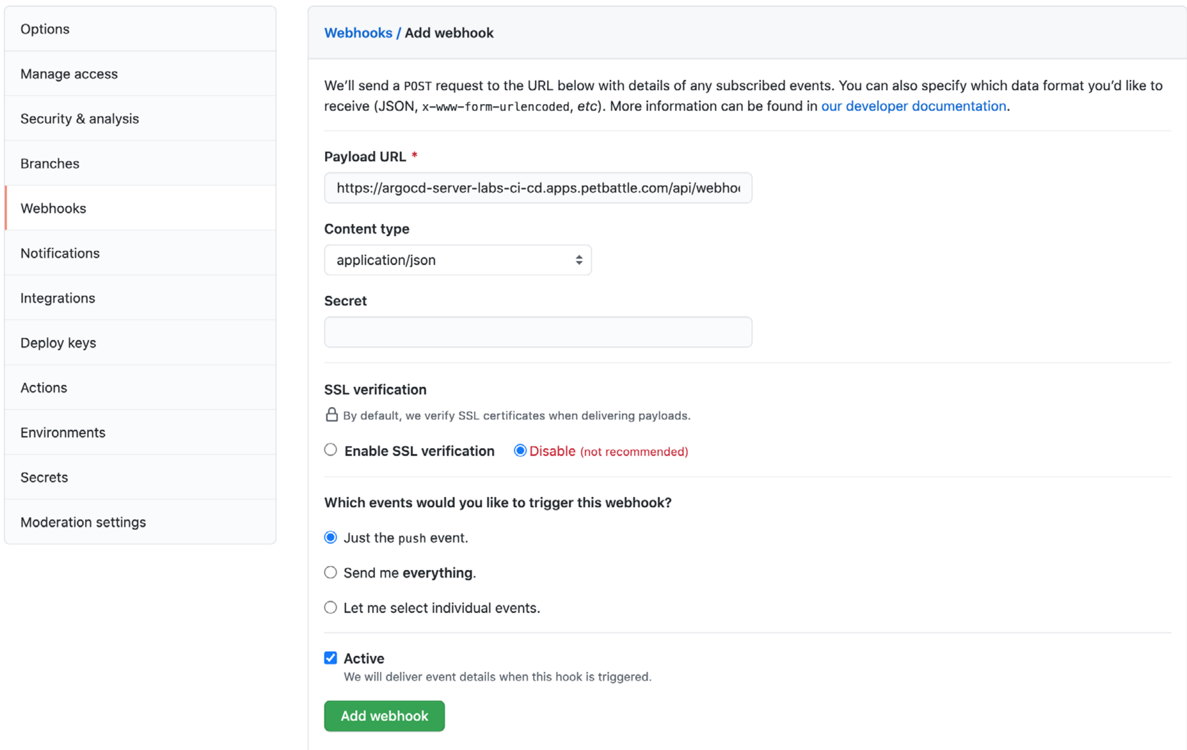


Figure 14.35: Webhook to trigger Argo CD on Git commit

Jenkins

Every time we commit to our source code repository, we want Jenkins to run a build. We're using the multibranch plugin for Jenkins, so this means that when we commit to the repository, the webhook will trigger a branch scan, which should bring back any new feature branches to build pipelines or create builds for any new code commits on any branch.

Configuring the Jenkins webhook for the `pet-battle` frontend is simple. On GitHub's *Hooks* page, add the URL to our Jenkins instance in the following form, where the trigger token is the name of our GitHub project. As a convention, I tend to use the name of the Git project as the token, so the same would apply for the backend if you were building it using Jenkins too:
`JENKINS_URL/multibranch-webhook-trigger/invoke?token=[Trigger token]`

For example, the frontend application's webhook URL would look something like this:

https://jenkins-labs-ci-cd.apps.petbattle.com/multibranch-webhook-trigger/invoke?token=pet-battle

Bringing It All Together

We have now gone through what a Jenkins file is and what it does for us. We've spoken about branching and what we mean for a build to be a release candidate (that is, a version bump and on master/main). We've touched on deploying using Helm and GitOps to commit our change and have Argo CD roll out the change for us…but how do we connect Jenkins up to all this magic?

As with all these things, there are several ways. We *could* open up Jenkins and hit **New Job > Multibranch Pipeline**, configure it to point to our Git repository, and set it to be triggered by a webhook, but that feels like the old way of doing things. I want a repeatable process so I don't have to do this step each time. Enter our `seed-multibranch-pipelines` job! Some of you may have noticed that Jenkins was configured to point to our organization's GitHub for PetBattle when we deployed the Helm chart from Ubiquitous Journey. We set some environment variables on the image (in `ubiquitous-journey/values-tooling.yaml`) to point to our GitHub organization as follows:

```
- name: GITHUB_ACCOUNT
value: 'petbattle'
- name: GITHUB_ORG
value: 'true'
```

If you're following along with a fork of the Ubiquitous Journey and want to see the pipeline run end to end, update both ARGOCD_CONFIG_REPO to point to your fork and QUAY_ACCOUNT to resolve to your user on Quay.io.

These are used by the `seed-multibranch-pipelines` job that is baked into the Jenkins image to scan the organization for repositories that contain a `Jenkinsfile` and are not archived. If it finds any, it will automatically scaffold out multibranch Jenkins jobs for us. In our case, we have a `Jenkinsfile` for both the Cats API and the PetBattle frontend, so jobs are created for us without having to configure anything! If you're following along and not using GitHub but GitLab, you can set `GITLAB_*` environment variables to achieve the same effect.

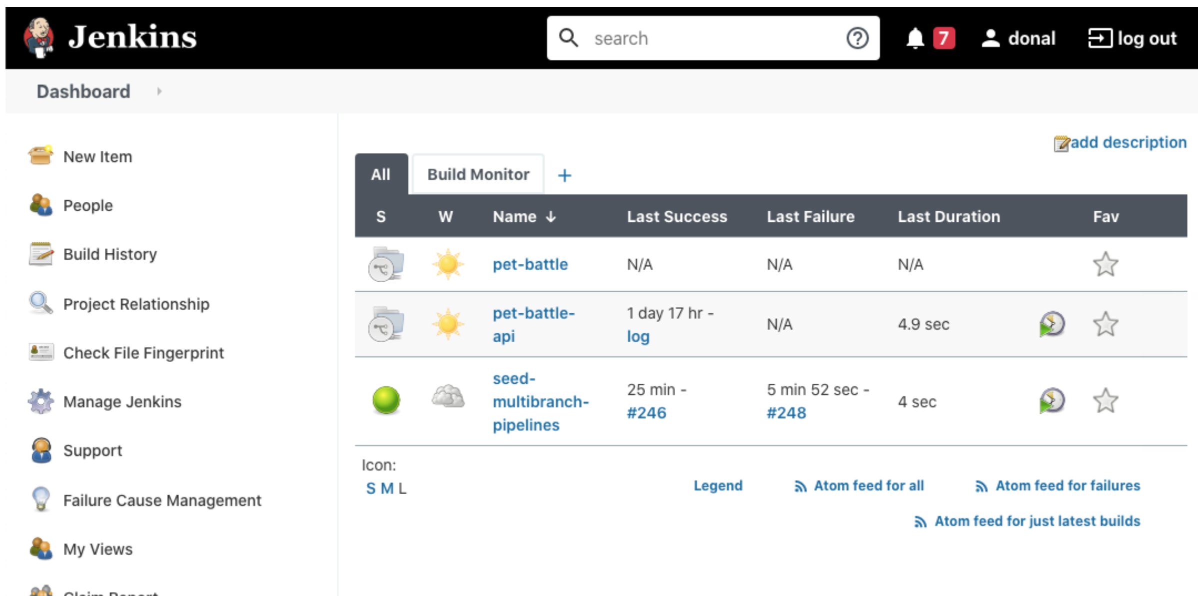


Figure 14.36: Jenkins seed to scaffold out our Jenkins jobs

If you open Jenkins and drill down into the `pet-battle` folder for the frontend code base, you should see builds; for example, a Git branch called `cool-new-cat` and the `master` with pipeline executions for each of them. Opening the Blue Ocean view, we get a much better understanding of the flow control we built, as previously discussed.

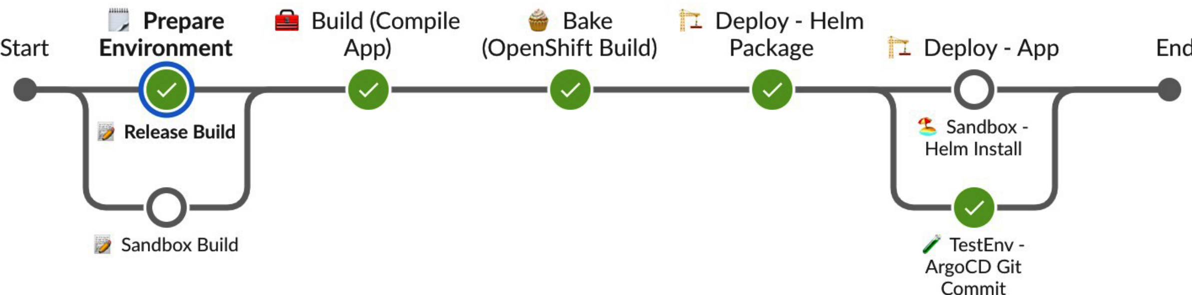


Figure 14.37: Jenkins release candidate pipeline

For the master branch, which we deem to be a release candidate, the artifacts that are built could go all the way. If we are updating our application, we bump the manifest version along with any changes we're bringing in and Git commit, which should trigger the build. From this point, our build environment is configured, and the pipeline should execute. We target an external repository and the image that's built will be pushed to Quay.io for portability across multiple clusters. Our Helm chart's values are patched and pushed to Nexus for storage. If we need to update our Helm chart itself, for example, to add some new configuration to the chart or add a new Kubernetes resource, we should of course bump the chart version too. For our deployment, we patch the Argo CD config repository (Ubiquitous Journey) with the new release information, and it should sync automatically for us, deploying our application to the `labs-test` namespace! We then run a verify step to check that the version being rolled out matches the new version (based on the labels) and has been successful.

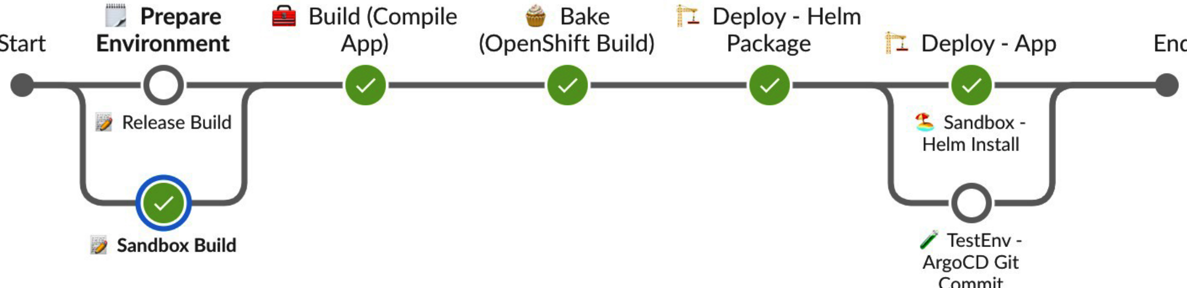


Figure 14.38: Jenkins feature development pipeline

For our feature branches, the idea is much the same, but without the need for an external repository. Our charts are also manipulated to override the name to include the branch. This means that on each commit to a feature branch, we get a new application deployed containing the branch name in the route. So, for our `cool-new-cat` branch, the application is deployed as `cool-new-cat-pet-battle` and is available in the developmental environment.

The remaining stages that were added to the Big Picture, System Test and Promote, will be covered in the next chapter, when we look in more detail at the testing for PetBattle.

What's Next for Jenkinsfile

Jenkins has been around for quite some time. It is not the most container-native approach to building software but there is a rich ecosystem surrounding it. It's lasted a long time because people like it! Hopefully, this gives you a taste of what can be done with Jenkins for our PetBattle applications, but it's by no means the end. There are a few plot holes in the story, as some of you may have noticed. For example, once a build has been successfully deployed to the test environment, how do I promote it onward? Should I do more testing? Well, the answer will come in the next chapter when we explore system tests and extend our pipeline further to include promoting images. At the end of a successful pipeline execution, the values file in our repository is not updated; we should be thinking about writing the successful build artifact details back to the repository, so it's always got a sensible default set to what is currently deployed.

The stages we have written here are fairly massive and do have some `bash` and other logic inside of them. If you were to build a non-frontend application, for example, you would want to build something in Golang. For the most part, the only thing that needs to change is the Build stage, as the act of putting something in a box, and how we package our Helm chart and deploy the app, remains the same. Once the app, in any language or framework, is in a container, then how we ship it remains the same. This means there is a high potential for reusing the code in the Bake and Deploy stages again and again, thus lowering the bar for adopting new technologies on a platform such as OpenShift. But be careful – copying and pasting the same steps across many jobs in a large estate of apps can lead to one mistake being copied around. Changes to the pipeline can become costly too, as you have to update each `Jenkinsfile` in each repository.

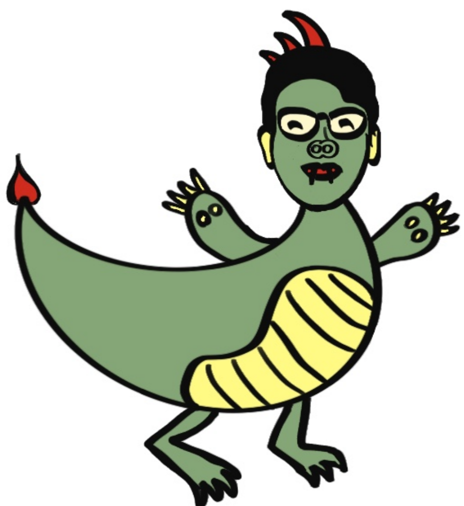


Figure 14.39: A lurking dragon, watch out!

Jenkins does tackle these problems with the use of shared libraries, and more recently the **Jenkins Templating Engine** (**JTE**). The JTE tackles the problem by enforcing pipeline approaches from a governance point of view. While this might seem like a great way to standardize across an enterprise—here be dragons!

Applying a standard pipeline without justification or the ability for teams to pull requests and make changes for their own specific use case is the same as having Dev and Ops in separate rooms. We've worked with plenty of customers who have tried approaches like this and ultimately it makes them go slower, rather than faster. The teams putting the pipelines in place think they're helping and providing a great service, but when things go wrong, they are the bottleneck to fixing it. For some teams, the hammer approach might not be applicable for their use case and so the pipeline becomes something in the way for them to go faster.

Tekton is another way for us to get greater pipeline reusability and also honor more of our GitOps landscape. Let's explore it now for our Java microservices.

Tekton–The Backend

Tekton[8] is an open source cloud-native CI/CD tool that forms the basis for OpenShift Pipelines.[9]

Tekton Basics

There are many similarities between what Jenkins does and what Tekton does. For example, both can be used to store pipeline definitions as code in a Git repository. Tekton is deployed as an operator in our cluster and allows users to define in YAML `Pipeline` and `Task` definitions. Tekton Hub[10] is a repository for sharing these YAML resources among the community, giving great reusability to standard workflows.

8 https://tekton.dev

9 https://docs.openshift.com/container-platform/4.7/cicd/pipelines/understanding-openshift-pipelines.html

10 https://hub.tekton.dev

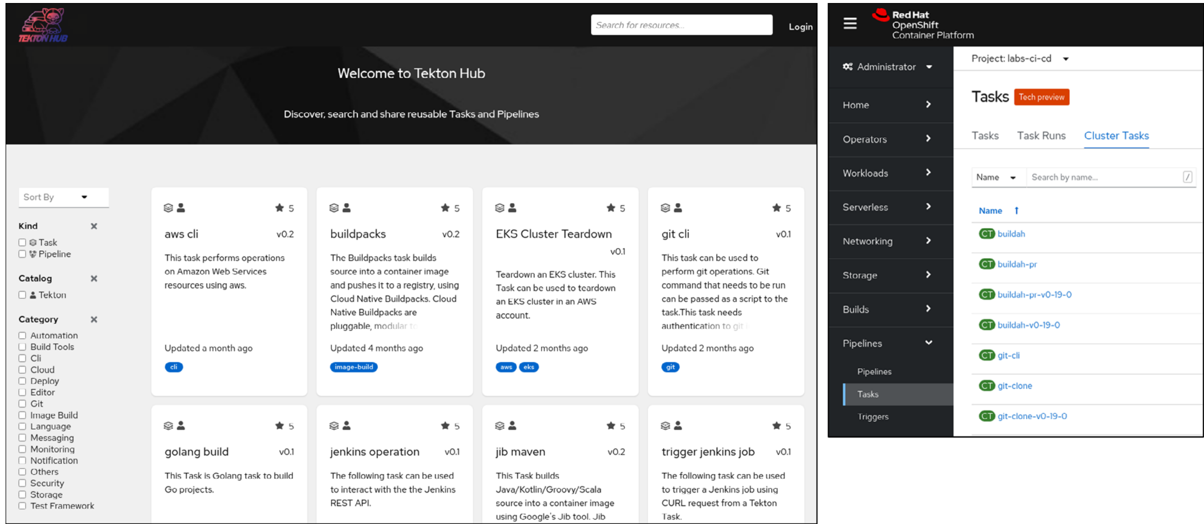


Figure 14.40: Tekton Hub and OpenShift Cluster tasks

OpenShift also makes these available globally as `ClusterTasks`. To write a pipeline you can wire together these task definitions. OpenShift provides a guided `Pipeline` builder UI for just this task. You link various tasks together and define parameters and outputs as specified in each task definition.

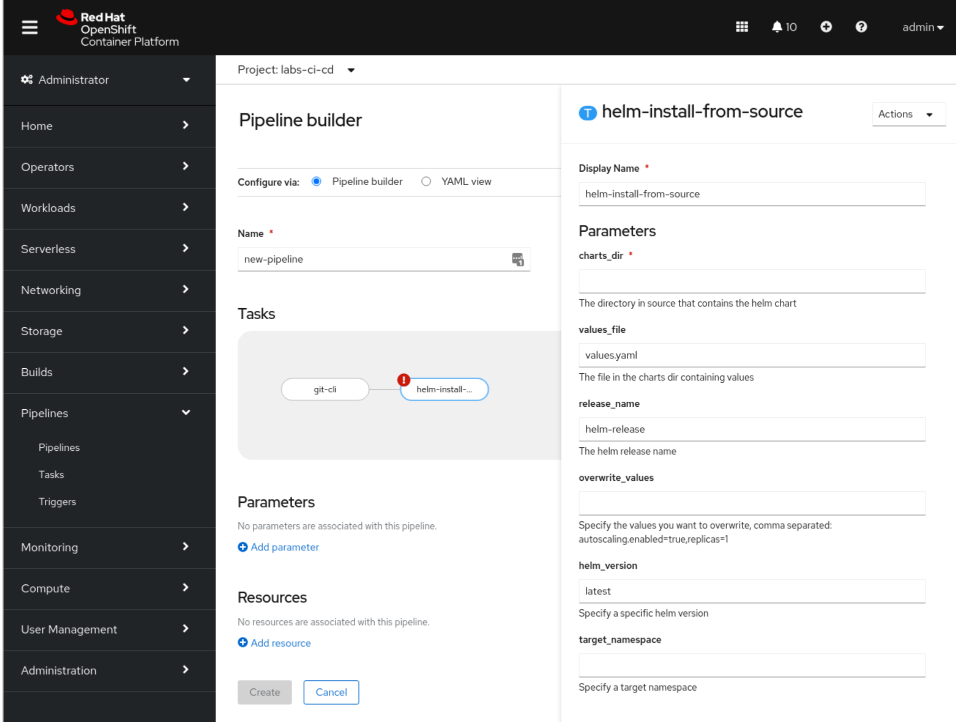


Figure 14.41: OpenShift Pipeline builder UI

There are numerous task activities in our pipeline definitions that require persistent storage. When building our backend PetBattle API and Tournament applications using Maven, we pull our Java dependencies via our Nexus repository manager. To speed up this process, we can perform the same caching we might do on our laptops and store these locally between builds in a `.m2/repository` folder and share this between builds. We also use persistent storage for built artifacts so they can be shared between different steps in our pipeline. Another use case is to mount Kubernetes secrets into our pipelines:

```
# maven pipeline
apiVersion: tekton.dev/v1beta1
kind: Pipeline
metadata:
  name: maven-pipeline
  labels:
    petbattle.app/uj: ubiquitous-journey
spec:
  workspaces:
    - name: shared-workspace
    - name: maven-settings
    - name: argocd-env-secret
    - name: maven-m2
    - name: git-auth-secret


# binding the workspace in the PipelineRun object
  resourcetemplates:
    - apiVersion: tekton.dev/v1beta1
      kind: PipelineRun
      metadata:
      ...
        workspaces:
        - name: shared-workspace
          persistentVolumeClaim:
            claimName: build-images
        - name: maven-settings
          persistentVolumeClaim:
            claimName: maven-source
        - name: argocd-env-secret
          secret:
            secretName: argocd-token
        - name: maven-m2
          persistentVolumeClaim:
```

```
            claimName: maven-m2
        - name: git-auth-secret
          secret:
            secretName: git-auth
```

In Tekton, we link these Kubernetes objects with the named `workspaces` when we create what is called the `PipelineRun`, a piece of code that represents one run of a pipeline. Similarly, the execution of a single task is a `TaskRun`. Each `workspace` is then made available for the tasks in that `PipelineRun` as shown.

Reusable Pipelines

There are some choices to be made before you start writing and designing your Tekton pipeline. The first is to choose whether you write a pipeline for each application, or whether you write reusable pipelines that can be used for applications that are similar.

In PetBattle, we started with one pipeline per application; this is similar to having a `Jenkinsfile` in each application Git repository. Both the API and Tournament PetBattle applications are built using Java, Quarkus, and Maven, so it makes sense to consolidate the pipeline code and write a reusable parameterized pipeline for these two applications because they will always have similar tasks. We use our `maven-pipeline` in PetBattle to do this.

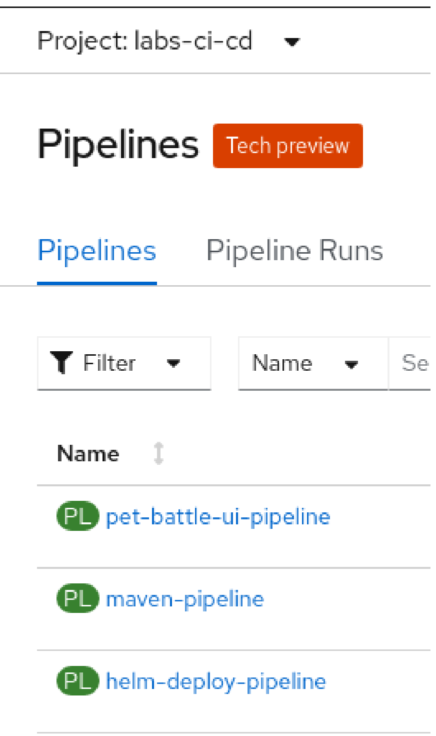


Figure 14.42: PetBattle's Tekton pipelines

Of course, you could keep the reuse to the Task level only but we share common tasks across the PetBattle UI, API, and Tournament applications. Ultimately, the development team has to balance the benefits of maintaining one pipeline over application pipeline autonomy. There is no one-size-fits-all answer.

Build, Bake, Deploy with Tekton

The next step is to start designing what we put into our pipeline. This is a very iterative process! In our Big Picture, we talked about the Build, Bake, and Deploy process, so it makes sense to add pipeline task steps that follow this methodology.

Tasks

- T git-clone (fetch-app-repository)
- T git-clone (fetch-cicd-repository)
- T maven (code-analysis)
- T sonarqube-quality-gate-check (quality-gate-check)
- T maven (maven-run)
- T openshift-kustomize (kustomize-build-config)
- T openshift-client (oc-start-build)
- T helm-upload-chart (upload-chart)
- T openshift-client (oc-tag-image-test)
- T openshift-client (oc-tag-image-stage)
- T helm-install-from-chartrepo (helm-install-apps-dev)
- T git-commit-argo-versions (git-commit-test)
- T helm-install-from-source (helm-argocd-apps-test)
- T argocd-sync-and-wait (argocd-sync-application-test)
- T git-commit-argo-versions (git-commit-stage)
- T helm-install-from-source (helm-argocd-apps-stage)
- T argocd-sync-and-wait (argocd-sync-application-stage)

Figure 14.43: The list of task definitions used by PetBattle's Tekton Pipelines

The `maven-pipeline` starts by cloning the application and CI/CD (Ubiquitous Journey) repositories into the shared `workspace`. We check the code quality by invoking Maven to build and test the application, with quality reports being uploaded to our SonarQube image.

We check that the quality gate in SonarQube has passed and then invoke Maven to package our application. Tekton offers us useful constructs to retry a task step if it fails by specifying the number of `retries` as well as the ordering of task steps using the `runAfter` task name list.

```
    - name: quality-gate-check
      retries: 1
      taskRef:
        name: sonarqube-quality-gate-check
      workspaces:
        - name: output
          workspace: shared-workspace
      params:
      - name: WORK_DIRECTORY
        value: "$(params.APPLICATION_NAME)/$(params.GIT_BRANCH)"
      runAfter:
      - save-test-results
```

In Java Quarkus, the packaging format could be a fat JAR, an exploded fast JAR, or a native GraalVM-based image. There are various trade-offs with each of these formats.[11] However, we are using the exploded fast JAR in PetBattle, which allows us to trade off between faster build times or faster startup times. This is the end of the Build stage. We have moved the unit testing left in our pipeline, so we get fast feedback on any code quality issues before we move on to the Bake and Deploy pipeline phases.

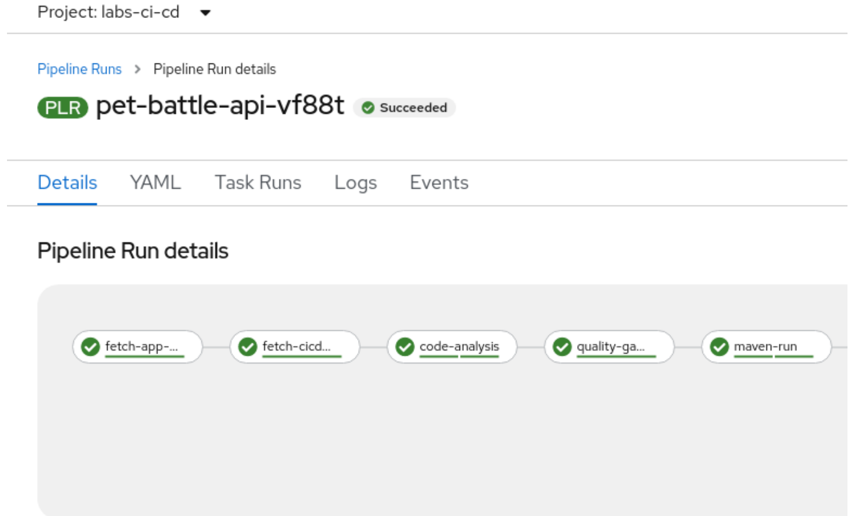


Figure 14.44: The view of a PipelineRun in OpenShift showing the tasks being executed

11 https://quarkus.io/guides/maven-tooling

The Bake stage is next. We use a standard OpenShift BuildConfig object, which is loaded using Kustomize, as we do not package that with our Helm chart. We perform a binary build using the `oc start build` command on the packaged application. We decided not to upload the built application package to Nexus because we want to work with container images as our unit of deployment. If we were building libraries that needed to support our services, then they should be captured in Nexus at this stage. It is worth pointing out that we could also push the image to an external registry at this point in time so it can be easily shared between OpenShift clusters.

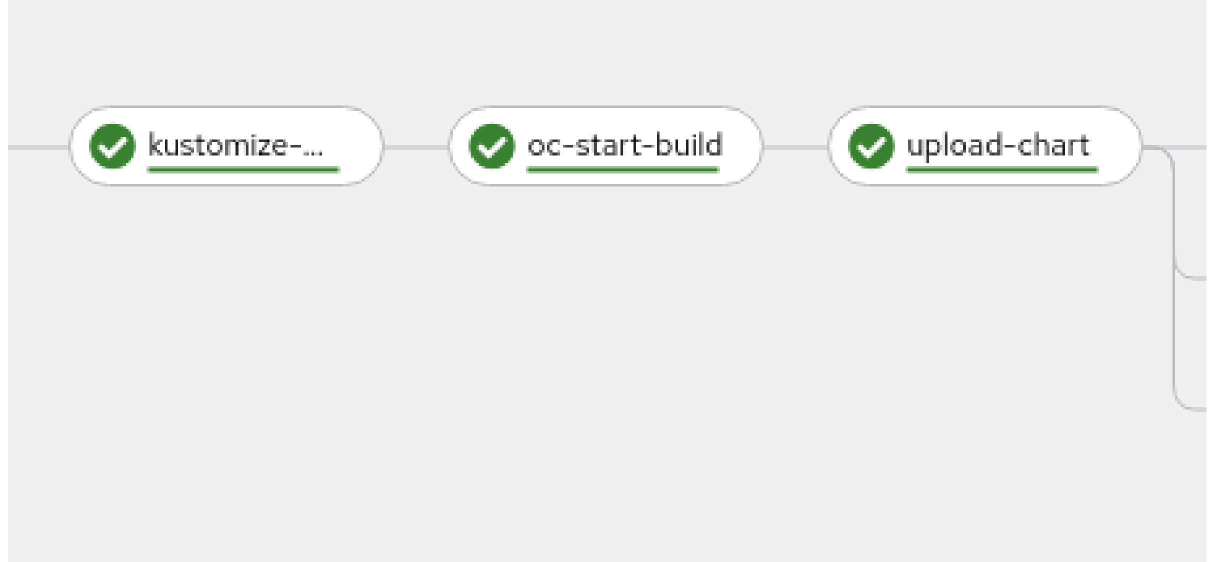


Figure 14.45: Bake part of the pipeline

The next step is to lint and package the application Helm chart. The versioned chart is then uploaded to Nexus. If we were on an application branch, the next pipeline step would be a `helm install` into the `labs-dev` project. We can make use of `when` statements in our Tekton pipeline to configure such behavior:

```
- name: helm-install-apps-dev # branches only deploy to dev
      when:
        - Input: "$(params.GIT_BRANCH)"
          Operator: notin
          Values: ["master","trunk","main"]
      taskRef:
        name: helm-install-from-chartrepo
```

When on trunk/HEAD, the ImageStream is versioned and tagged into the namespaces we are going to deploy our application to (`labs-test`, `labs-staging`). Because we are practicing GitOps, the applications are deployed using Argo CD and Git. The Argo CD app-of-apps values files are updated with the new chart and image versions. This is checked into source code by the pipeline and `git commit` is executed. Argo CD is configured to automatically sync our applications in `labs-test` and `labs-staging`, and the last step of the pipeline is to make sure the sync task was successful.

Figure 14.46: Deploy part of the pipeline

There is a lot of pipeline information available to the developer in the OpenShift web console and all of the pipeline task logs can be easily seen.

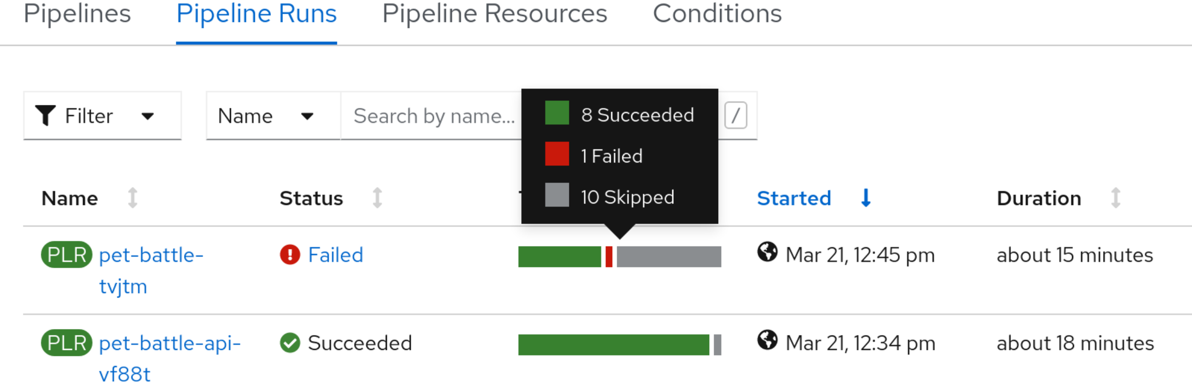


Figure 14.47: Tekton pipeline progress and status hover

Tekton also has a great command-line tool called `tkn`, which can be used to perform all of the pipeline actions available in the OpenShift console, such as viewing logs, starting pipeline runs, and defining Tekton objects.

```
$ tkn pr list -n labs-ci-cd
```

```
NAME                          STARTED    DURATION     STATUS
pet-battle-tournament-44vg5   1 days ago 27 minutes   Failed
pet-battle-9d9c7              2 days ago 19 minutes   Succeeded
pet-battle-api-jcgn2          2 days ago 21 minutes   Succeeded
pet-battle-kfch9              2 days ago 5 minutes    Failed
pet-battle-tournament-br5xd   2 days ago 24 minutes   Failed
pet-battle-api-5l4sd          2 days ago 23 minutes   Failed
```

Let's now take a look at how we can trigger a build.

Triggers and Webhooks

On every developer push to Git, we wish to trigger a build. This ensures we get the fastest feedback for all of our code changes. In Tekton this is achieved by using an `EventListener` pod object. When created, a pod is deployed, exposing our defined trigger actions.

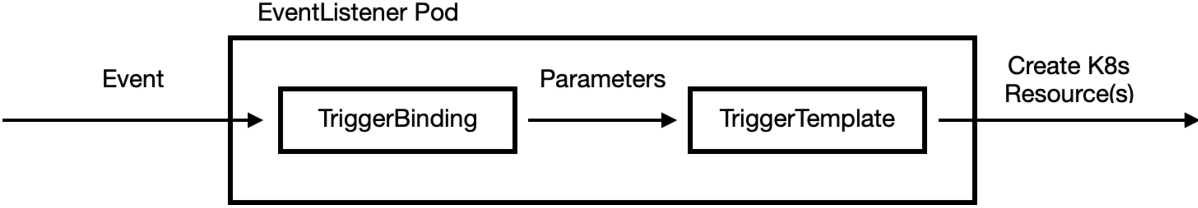


Figure 14.48: Tekton Triggers flow

Tekton Triggers work by having `EventListener` objects receive incoming webhook notifications, processing them using an interceptor, and creating Kubernetes resources from templates if the interceptor allows it, with the extraction of fields from the body of the webhook (there's an assumption that the body is a JSON file):

```
apiVersion: triggers.tekton.dev/v1alpha1
kind: EventListener
metadata:
  name: github-webhook
  labels:
    app: github
spec:
  serviceAccountName: pipeline
  triggers:
    - name: pet-battle-api-webhook-all-branches ...
    - name: pet-battle-api-webhook-pr ...
    - name: pet-battle-tournament-webhook-all-branches ...
    - name: pet-battle-tournament-webhook-pr ...
    - name: pet-battle-webhook-all-branches ...
    - name: pet-battle-webhook-pr ...
```

In OpenShift, we expose the EventListener webhook endpoint as a route so that it can be wired into Git. Different types of **source control managers** (**SCMs**) define different webhook payloads. We are using GitHub, so it is those webhook payloads[12] that we need to use to help define the parameters used to create the TriggerBinding and pass to our TriggerTemplate. The TriggerTemplate then defines the Tekton resources to create. In our case, this is a PipelineRun or TaskRun definition.

```
triggers:
    - name: pet-battle-api-webhook-all-branches
      interceptors: # fixme add secret.ref
        - cel:
            filter: >-
              (header.match('X-GitHub-Event', 'push') &&
                body.repository.full_name ==
                                        'petbattle/pet-battle-api')
            overlays:
            - key: truncated_sha
              expression: "body.head_commit.id.truncate(7)"
            - key: branch_name
              expression: "body.ref.split('/')[2]"
            - key: app_of_apps_key
              expression: "body.repository.name.replace('-','_',-1)"
      bindings:
      - kind: TriggerBinding
        ref: github-trigger-binding
      template:
        ref: pet-battle-api-maven-trigger-template
```

Tekton uses an expression language, known as the **Common Expression Language** (**CEL**),[13] to parse and filter requests based on JSON bodies and request headers. This is necessary because of the differing webhook payloads and potentially different Git workflows. For example, we are using GitHub and treat a pull request differently from changes to our main/HEAD. One customization we make that you can see above is to define the Argo CD app-of-apps key in the trigger binding based on the Git repository name. This allows us to check the synchronization of just the one application that changed and not the whole application suite during the Deploy phase of our pipeline. While triggering seems complex, the flexibility is required when dealing with all the various Git SCMs and workflows that are available to development teams.

---

12 https://docs.github.com/en/developers/webhooks-and-events/webhook-events-and-payloads

13 https://github.com/google/cel-go

There are some convenience templates loaded into the `labs-ci-cd` project by Ubiquitous Journey that can be used to manually trigger a `PipelineRun`—this is handy if you have not configured the GitHub webhook yet.

```
$ oc -n labs-ci-cd process pet-battle-api | oc -n labs-ci-cd create -f-
$ oc -n labs-ci-cd process pet-battle | oc -n labs-ci-cd create -f-
$ oc -n labs-ci-cd process pet-battle-tournament | oc -n labs-ci-cd create -f-
```

You can manually add webhooks to your GitHub projects[14] that point to the `EventListener` route exposed in the `labs-ci-cd` project.

```
$ oc -n labs-ci-cd get route webhook \
     -o custom-columns=ROUTE:.spec.host --no-headers
```

Otherwise, check out the PetBattle Ubiquitous Journey documentation for Tekton tasks that can be run to automatically add these webhooks to your Git repositories.

GitOps our Pipelines

Our pipeline, task, trigger, workspace, and volume definitions are themselves applied to our `labs-ci-cd` project using GitOps. The idea here is to minimize how hard it is to adapt our pipelines. We may want to add some more security checks into our pipeline steps, for example. If there are testing failures, or even service failures in production, then we need to adapt our pipelines to cater for further quality controls or testing steps. Adding new tools or modifying task steps becomes nothing more than pushing the pipeline as code definitions to Git.

```
# PetBattle Tekton objects
  - name: tekton-pipelines
    destination: labs-ci-cd
    enabled: true
    source: https://github.com/petbattle/ubiquitous-journey.git
    source_path: tekton
    source_ref: main
    sync_policy: *sync_policy_true
    no_helm: true
```

Within our Tekton source folder, we use Kustomize to apply all of the YAML files that define our Tekton objects. These pipeline objects are kept in sync by Argo CD.

14 https://docs.github.com/en/developers/webhooks-and-events/creating-webhooks

Which One Should I Use?

The CI/CD tooling landscape is massive[15] and also extremely vibrant and healthy. The CNCF landscape for tools in this category has no less than 36 products and projects today. In trying to answer the question of which one you should use; it is best to consider multiple factors:

- Does your team have previous skills in a certain tooling or language? For example, pipelines as code in Jenkins use the Groovy language, so if your team has Groovy or JavaScript skills, this could be a good choice.
- Does the tool integrate with the platform easily? Most of the tools in CNCF have good integration with Kubernetes already and have a cloud-native pedigree. That does not mean that all tools are the same in terms of deployment, platform integration, or lifecycle management–some may be **Software as a Service** (**SaaS**)-only offerings with agents, whereas some can be deployed per team using namespace isolation on your cluster. Others, such as Argo CD and Tekton, can be deployed at cluster scope using the operator pattern, and have their lifecycle managed via the **Operator Lifecycle Manager** (**OLM**). Tekton has great web console integration with OpenShift because of the OpenShift Pipelines operator.
- Tool deployment model: Jenkins and Argo CD both have a client-server model for deployment. This can be problematic at larger scales, such as when looking after thousands of pipelines or hundreds of applications. It may be necessary to use multiple deployments to scale across teams and clusters. Argo CD and Tekton extend Kubernetes using CRDs and operator patterns, so deployment is more Kubernetes-native in its scaling model.
- Enterprise support: Most, but not all, the tools have vendor support. This is important for enterprise organizations that need a relationship with a vendor to cover certification, training, security fixes, and product lifecycles.

15 https://landscape.cncf.io/card-mode?category=continuous-integration-delivery&grouping=category

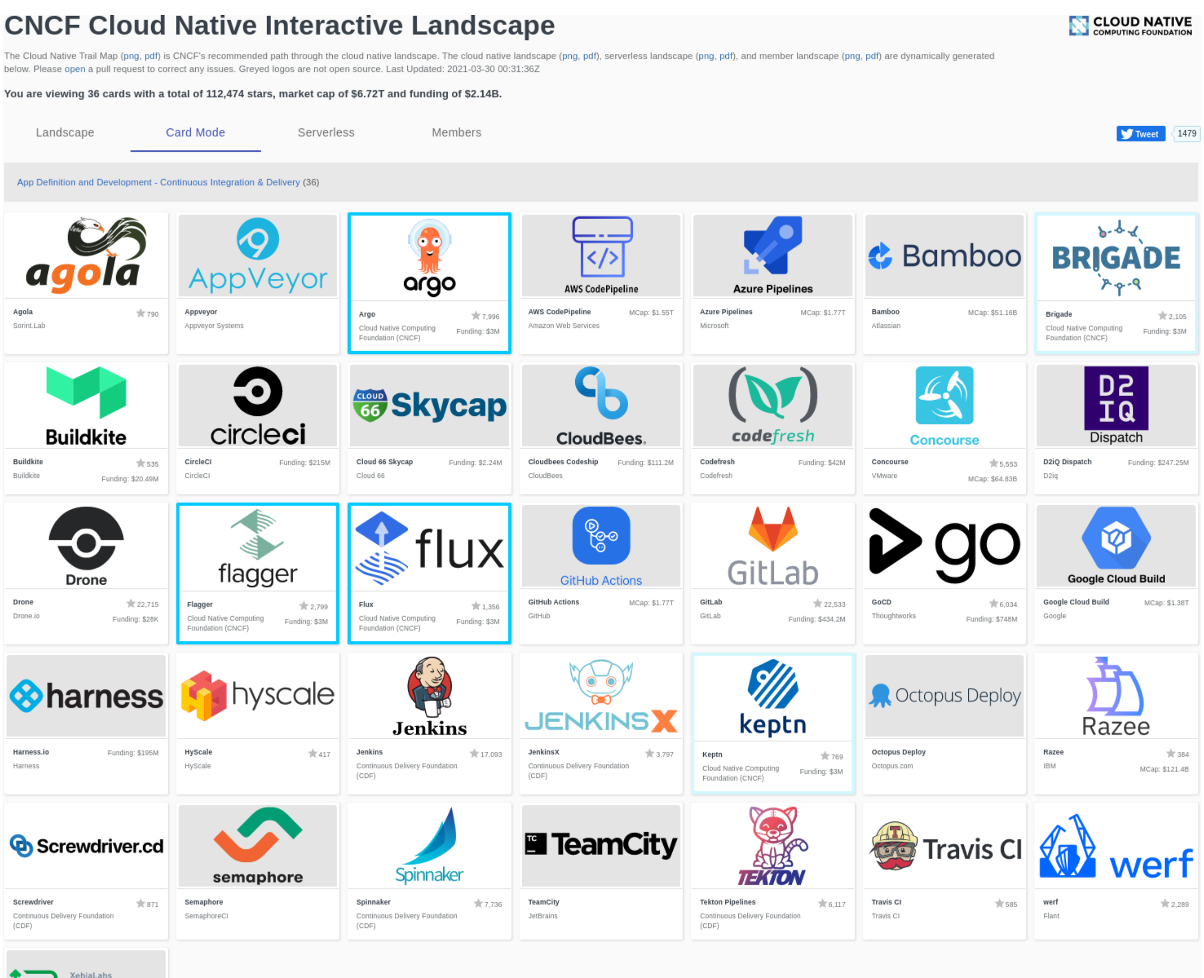


Figure 14.49: CNCF CI/CD tooling landscape

- Active open-source communities: A vibrant upstream community is important as a place to collaborate and share code and knowledge. Rapid development of features and plugins often occurs in a community based on real user problems and requests.
- One tool for both CI and CD or use different tools? As we have shown with PetBattle, sometimes it makes sense for CI to be a push-type model, and CD to be a pull-type model using different tools.

- Extensibility model: This is important for the ecosystem around the tooling. Jenkins has a great plugin model that allows lots of different extensions to the core. Tekton has a similar model, but it is different in that users have the ability to use any container in a task. It is important to weigh up these extensions as they offer a lot of value on top of the core tool itself. A good example is that Tekton does not manage test dashboards and results as well as Jenkins and its plugins do, so we might lean on Allure to do this. Reporting and dashboarding extensions are important to make the feedback loop as short as possible during CI/CD.

Once you have considered a few of these ideals, hopefully you will agree on the right set of tools for your product and team. A measure of design and planning is required to answer the question of where the various steps in your continuous deployment happen and what application packaging approach should be used (templated or not templated, for example). By now, we have instilled an experiment-driven approach to answering these types of questions, where it is not one or the other tool, but about choosing the right tool for the job at hand!

Conclusion

In this chapter we introduced how we use Git as the single source of truth. We covered taking our source code and packaging it using either

Tekton or Jenkins. In the next chapter, we will focus on testing, introducing a new component to our app using Knative, running A/B tests, and capturing user metrics using some of the advanced deployment capabilities within OpenShift.

15

Run It

There is a saying that *your code has no value until it runs in production*. The sentiment here is that until your customers use your software, it's of limited value for your business or organization. It is certainly a broad generalization! However, it does speak to the essential nature of software that its utility is directly related to being able to run it for whatever purposes it was ultimately written for. To reach production with the quality of service that our customers expect, all of the code must be put through its paces.

In this chapter, we are going to explore how the PetBattle team tests their software so they have greater confidence in its ability to run as expected in production. Testing is multifaceted, as we discussed in *Chapter 7*, *Open Technical Practices – The Midpoint*, and we are going to cover in some detail the types and scope of testing, from unit tests to end-to-end testing, through to security checks and more.

When the hobbyist version of the application went live, the PetBattle founders soon discovered that malicious content was being uploaded to the site. As part of this chapter, we'll look at a modern-day solution to this problem using a trained AI-ML model.

In the last section of this chapter, we explore some common cloud deployment patterns and demonstrate A/B testing and experimentation, for gaining insight into how we can safely measure and learn the impact of deploying new features in production.

The Not Safe For Families (NSFF) Component

As we mentioned earlier, one of the major issues that we faced when running the first generation of PetBattle was online trolls uploading inappropriate images to the system. This added to the operational overhead of running the platform because the PetBattle founders would have to search MongoDB for the offending images and remove them by hand—very tedious!

Ever innovating, the team decided to try and come up with an automated solution to this problem. One approach we decided to investigate was to use **artificial intelligence (AI)** to perform image classification on the uploaded images and incorporate this into the platform.

The field of AI in itself is a fascinating area of expertise that we won't even slightly go into here, other than to say that we are using a pre-trained image classification model served by the open source TensorFlow machine learning platform.

Great, but how do we go about running this on OpenShift?

The plan is to:

1. Generate or obtain a pre-trained image classification model.
2. Build containers containing the TensorFlow serving component that can serve up the model and make predictions based on our uploaded images.
3. Deploy and run the container on OpenShift in a "scale to zero" deployment model, aka Serverless.

Why Serverless?

When deploying a container on a Kubernetes-based platform, such as OpenShift, Kubernetes takes on the responsibility of managing the running container and, by default, restarting it if it terminates due to an error. Basically, there's always a container running. This is all good and fine for containers that are constantly receiving and processing traffic, but it's a waste of system resources constantly running a container that receives traffic either occasionally or in bursts.

What we'd like to achieve is to deploy a container and have it start up only when needed, that is, during incoming requests. Once active, we want it to process the incoming requests and then, after a period of no traffic, shut down gracefully until further incoming requests are received. We'd also like the container instances to scale up in the event of a surge of incoming requests.

It is possible to automate the scaling up and down of the number of container instances running on the platform using the Kubernetes Horizontal Pod Autoscaler; however, this does not scale to zero. We could also use something like the `oc scale` command, but this requires a fair amount of scripting and component integration. Thankfully, the Kubernetes community thought about this and came up with a solution called Knative.[1]

Knative has two major components, **Knative Serving** and **Knative Eventing**. Serving is used to spin up (and down) containers depending on HTTP traffic. Knative Eventing is somewhat equivalent but is focused on spinning up containers based on events and addresses broader use cases. For the purposes of this book, we are going to focus on using Knative Serving. However, we will also give an example of how Knative Eventing could be used.

Generating or Obtaining a Pre-trained Model

We had been experimenting with image classification for a while. We started using some of the components from the Open Data Hub community (https://opendatahub.io/) and trained out models on top of pre-existing open source models that were available. We eventually generated a trained data model that could classify images that we deemed NSFF based on an implementation of Yahoo's Open NSFW Classifier,[2] which was rewritten with TensorFlow. While it was not perfect, it was a good enough model to start with.

A common pattern in the data science community is to serve up trained data models using tools such as Seldon,[3] which are part of Open Data Hub. For our purposes though, a simple object storage tool was all that was required. So, we chose MinIO,[4] a Kubernetes native object store. We decided we could scale that out later if needed, using more advanced storage mechanisms, for example, OpenShift Container Storage or AWS S3.

1 https://knative.dev/

2 https://github.com/yahoo/open_nsfw

3 https://www.seldon.io/

4 https://min.io/

We loaded the trained data model into MinIO and it looked as follows:

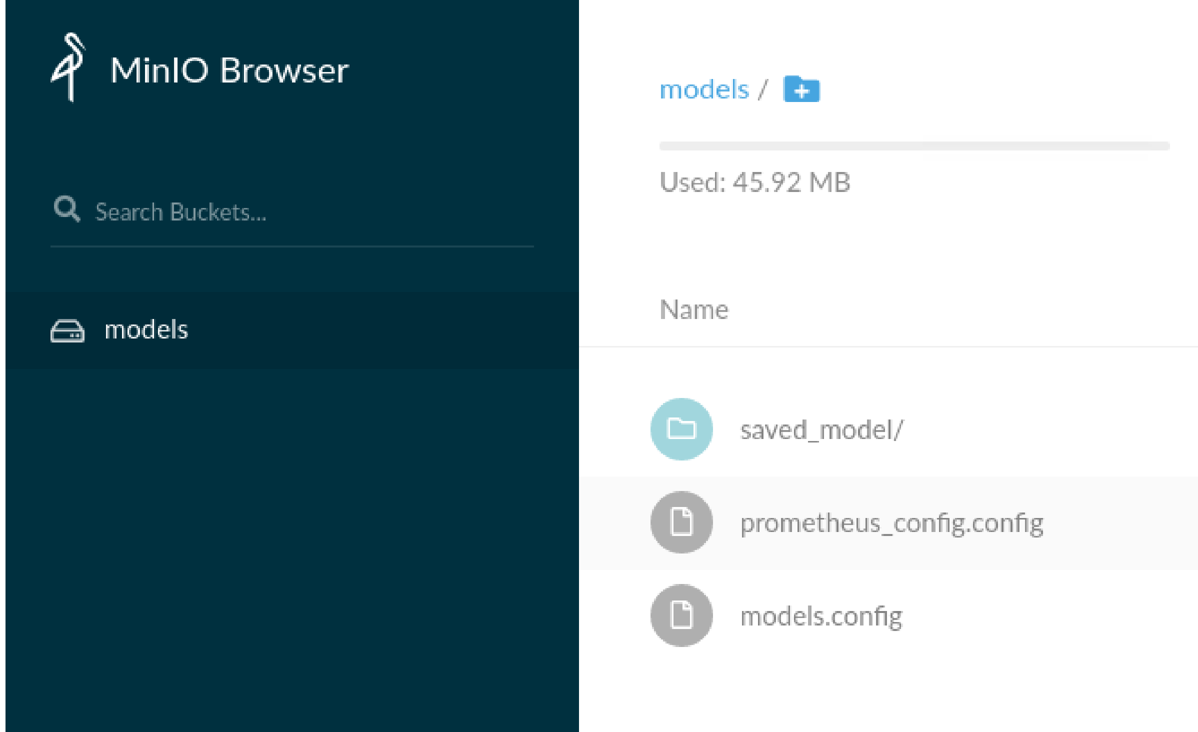


Figure 15.1: TensorFlow data model saved in MinIO

The saved model is something we can serve up using TensorFlow Serving,[5] which basically gives us an API endpoint to call our saved model with. There is an open source TensorFlow serving image we can deploy and it's a matter of configuring that to find our saved model in our S3 storage location.

We have glossed over the large portion of engineering that goes into making AI, ML, and Ops pipelines not because it is not an interesting subject, but mainly because it would require a whole other book to do it justice! If this subject is close to your heart, then take a look at the Open Data Hub project.[6] This is an open source project based on Kubeflow,[7] providing tools and techniques for building and running AI and ML workloads on OpenShift.

5 https://www.tensorflow.org/tfx/guide/serving

6 http://opendatahub.io/

7 https://www.kubeflow.org/

The OpenShift Serverless Operator

Before we start deploying our application software for the NSFF service, we need to add the OpenShift Serverless Operator[8] to our PetBattle Bootstrap. The operator is installed at the cluster scope so that any project that wants to use the Knative components Knative Serving and Knative Eventing may do so.

Let's use GitOps, ArgoCD, and Kustomize to configure and install the serverless operator. First, we can test out the configuration with ArgoCD. Log in using ArgoCD from the command line. Add the Git repository that contains the Knative serverless operator YAML subscription and create the application:

```
# Login to ArgoCD
$ argocd login $(oc get route argocd-server --template='{{ .spec.host }}' \
-n labs-ci-cd):443 --sso --insecure

# Add our repository
$ argocd repo add \
  https://github.com/rht-labs/refactored-adventure.git

# Create the Knative Operator - this may have already been created for you
but here is how to do it on the command line.
# Create the Knative Operator
$ argocd app create knative\
  --repo https://github.com/rht-labs/refactored-adventure.git \
  --path knative/base \
  --dest-server https://kubernetes.default.svc \
  --dest-namespace openshift-serverless \
  --revision master \
  --sync-policy automated
```

8 https://github.com/openshift-knative/serverless-operator

Once installed, you should be able to see this installed successfully in the `openshift-serverless` namespace:

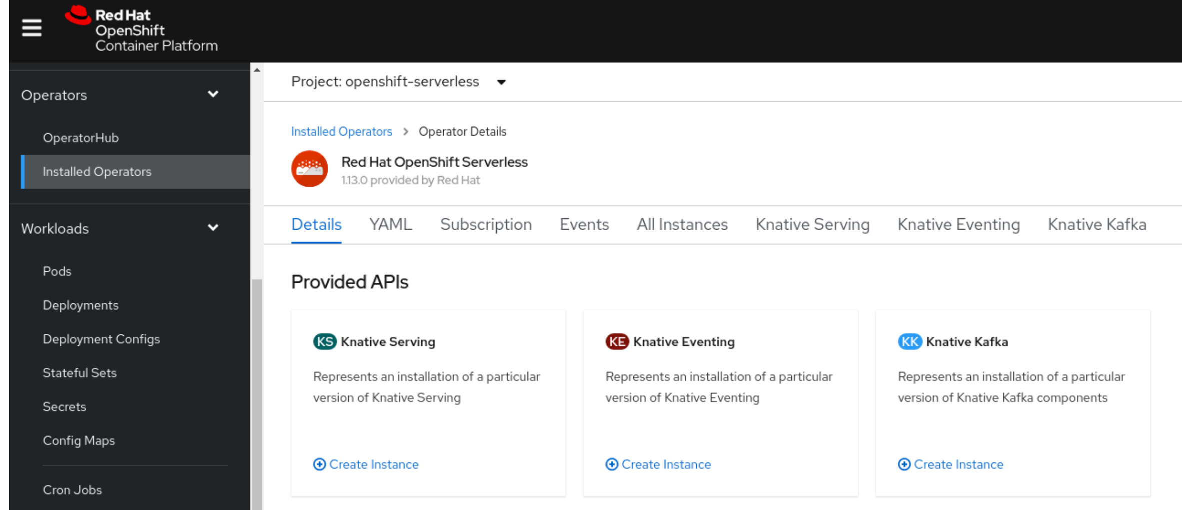


Figure 15.2: The OpenShift Serverless Operator (Knative)

We can also put this in our PetBattle UJ bootstrap from *Chapter 7*, *Open Technical Practices – The Midpoint*, so that we don't need to run these commands manually. Add the following to our `values-tooling.yaml` and check it into Git:

```
# Knative stanza in values-tooling.yaml
- name: knative
    enabled: true
    destination: openshift-serverless
    source: https://github.com/rht-labs/refactored-adventure
    source_path: knative/base
    source_ref: master
    sync_policy: *sync_policy_true
    no_helm: true
```

The operator is now ready for us to use to deploy our Knative service.

Deploying Knative Serving Services

There are a few ways in which to create Knative Serving services. We can create the Knative service definition and install that into our cluster. We have packaged this up as a Helm chart for easy installation:

```
$ helm upgrade --install pet-battle-nsff \
petbattle/pet-battle-nsff \
--version=0.0.2 \
--namespace petbattle
```

It may take a minute or so for the containers to start up and load the model data into MinIO; they may restart a few times while doing this. The output of the `oc get pods` command should look like this once successful – the MinIO S3 pod and its completed data load and a TensorFlow Knative service pod:

```
$ oc get pods --namespace petbattle

NAME                                     READY   STATUS    RESTARTS  AGE
Minio-pet-battle-nsff-594fc7759-j7lwv    1/1     Running     0       88s
minio-pet-battle-nsff-dataload-7x8jg     0/1     Completed   2       88s
minio-pet-battle-nsff-gfjhz              0/1     Completed   3       88s
tensorflowserving-pet...-7f79956d9qfp  2/2     Running     2       85s
```

After a couple of minutes, the Knative Serving TensorFlow pod will terminate because it is not yet being called. This is what's called Serverless scale to zero, that is, when there are no calling workloads there is no need to run the service. An equivalent service can also be created using the Knative command-line tool **kn**, which can be downloaded and installed from the OpenShift[9] console. This is useful if you want to create a new service or are developing a service from scratch:

```
$ kn service create tensorflowserving-pb-nsff --namespace petbattle \
  --image=docker.io/tensorflow/serving:latest \
  --cmd "tensorflow_model_server" \
  --arg "--model_config_file=s3://models/models.config" \
  --arg "--monitoring_config_file=s3://models/prometheus_config.config" \
  --arg "--rest_api_port=8501" \
  --env S3_LOCATION=minio-pet-battle-nsff:9000 \
  --env AWS_ACCESS_KEY_ID=minio \
  --env AWS_SECRET_ACCESS_KEY=minio123 \
  --env AWS_REGION=us-east-1 \
  --env S3_REGION=us-east-1 \
  --env S3_ENDPOINT=minio-pet-battle-nsff:9000 \
  --env S3_USE_HTTPS="0" \
  --env S3_VERIFY_SSL="0" \
  --env AWS_LOG_LEVEL="3" \
  --port 8501 \
  --autoscale-window "120s"
```

9 https://docs.openshift.com/container-platform/4.7/serverless/serverless-getting-started.html

Here, we use command-line arguments and environment variables to tell the TensorFlow serving image how to run. The `--image` field specifies the container image and version we wish to run – in this case, the latest TensorFlow serving image. The `--cmd` field specifies the binary in the image we wish to run, for example, the model server command `tensorflow_model_server`. The `--arg` and `--env` variables specify the configuration. The trained model is served from the `S3 minio` service so we specify how to access the S3 endpoint. There are many configurations available to Knative Serving, such as autoscaling global defaults, metrics, and tracing. The `--autoscale-window` defines the amount of data that the autoscaler takes into account when scaling, so in this case, if there has been no traffic for two minutes, scale the pod to 0.

The Knative website[10] goes into a lot more detail about the serving resources that are created when using Knative Serving and the configuration of these. To find the URL for our service, we can use this command:

```
$ kn route list
```

This gives us the HTTP URL endpoint to test our service with. It is worth noting that we can have multiple revisions of a service and that within the route mentioned previously, we can load balance traffic across multiple revisions. The following diagram depicts how this works in practice:

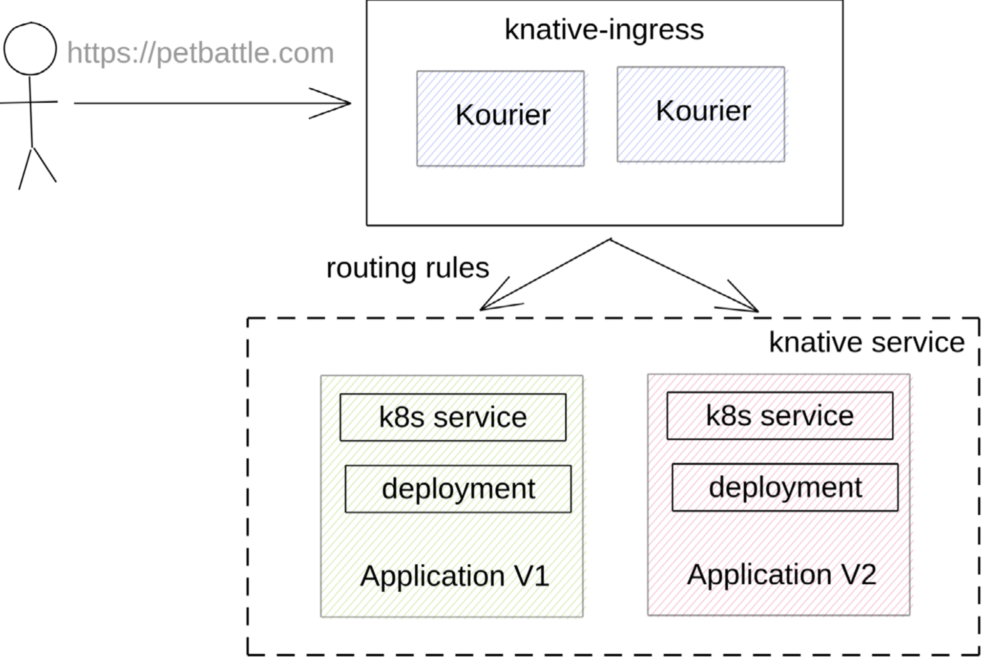


Figure 15.3: Knative routing for multiple application revisions

10 https://knative.dev/docs/serving/

Kourier[11] is a lightweight ingress router based on an Envoy gateway. Using Knative service configuration, a user can specify routing rules that Knative Serving applies. This can be very useful when we're experimenting with different AI models or wanting to do A/B, Blue/Green, or Canary-type deployments, for example.[12]

Invoking the NSFF Component

A simple HTTP GET request on the route is all that's required to invoke the component. The pod spins up and services the request usually within a couple of seconds and then spins down after a period of time (the `--autoscale-window` specified in the `kn` command-line argument, that is, 120 seconds). Using the output from the `kn list route` command, let's check if the AI TensorFlow model is available. The `state` should read `AVAILABLE`:

```
$ curl <url from kn route list>/v1/models/test_model
# For example

$ curl http://tensorflowserving-pet-battle-nsff-labs-
dev.apps.hivec.sandbox882.opentlc.com/v1/models/test_model
{
 "model_version_status": [
  {
   "version": "1",
   "state": "AVAILABLE",
   "status": {
    "error_code": "OK",
    "error_message": ""
   }
  }
 ]
}
```

11 https://developers.redhat.com/blog/2020/06/30/kourier-a-lightweight-knative-serving-ingress/

12 https://medium.com/@kamesh_sampath/serverless-blue-green-and-canary-with-knative-kn-ad49e8b6aa54

We should also see a pod spinning up to serve the request, using:

```
$ oc get pods \
-l serving.knative.dev/configuration=tensorflowserving-pet-battle-nsff \
--namespace petbattle

NAME                          READY   STATUS    RESTARTS   AGE
tensorflowserving-pet…        2/2     Running   0          21s
```

It then scales down to 0 after two minutes.

We want to test that our NSFF service works by sending it some images. We have two test sample images that have been encoded so they can be uploaded to the NSFF service.

- Safe for Families - Daisy Cat

- Not Safe for Families - Boxing

Figure 15.4: NSFF test images

Let's download these images for testing:

```
$ wget https://raw.githubusercontent.com/petbattle/pet-battle-nsff/
main/requests/tfserving/nsff-negative.json
$ wget https://raw.githubusercontent.com/petbattle/pet-battle-nsff/
main/requests/tfserving/nsff-positive.json
```

Now submit these to our NSFF service using a simple `curl` command:

```
$ HOST=$(kn service describe tensorflowserving-pb-nsff -o url)/v1/models/
test_model:predict

# Daisy Cat - Safe for Families
curl -s -k -H 'Content-Type: application/json \
  -H 'cache-control: no-cache' \
  -H 'Accept: application/json' \
  -X POST --data-binary '@nsff-negative.json' $HOST

{ "predictions": [[0.992712617, 0.00728740077]] }
# Not Safe For Families - Boxing
curl -s -k -H 'Content-Type: application/json' \
  -H 'cache-control: no-cache' \
  -H 'Accept: application/json' \
  -X POST --data-binary '@nsff-positive.json' $HOST

{ "predictions": [[0.30739361, 0.69260639]] }
```

The response from our model is a `predictions` array containing two numbers. The first is a measure of **Safe for Families**, the second is a measure of **Not Safe for Families**, and they add up to 1.

So, we can see that Daisy Cat has a very high safe for families rating (0.993) compared to our wrestlers (0.014) and we can use this in our PetBattle API to determine whether any given image is safe to display. By arbitrary testing, we have set a limit of >=0.6 for images we think are safe to view in the PetBattle UI.

We can redeploy our PetBattle API service to call out to the NSFF service by setting the `nssf.enabled` feature flag to `true` and using the hostname from the Knative service from a bash shell using the command line:

```
$ HOST=$(kn service describe tensorflowserving-pet-battle-nsff -o url)
$ helm upgrade --install pet-battle-api petbattle/pet-battle-api \
--version=1.0.8 \
--set nsff.enabled=true \
--set nsff.apiHost=${HOST##http://} \
--set nsff.apiPort=80 --namespace petbattle
```

If we now upload these test images to PetBattle via the UI and check the API server, we can see that the boxing picture has a **false** value for the `ISSFF` (Is Safe for Families) flag and Daisy Cat has a **true** value:

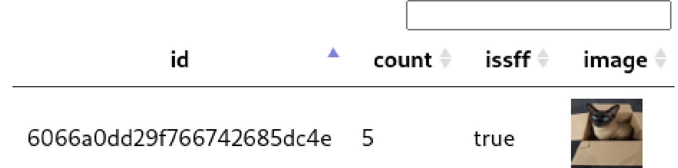


Figure 15.5: PetBattle API saved images with the ISSFF flag

The API code will not return any pictures to the PetBattle UI that are deemed NSFF. For example, the API code to return all pets in the PetBattle database is filtered by the `ISSFF` flag being set to `true`:

```
@GET
@Operation(operationId = "list",
           summary = "get all cats",
           description = "This operation retrieves all cats from the
                         database that are safe for work",
           deprecated = false, hidden = false)
public Uni<List<Cat>> list() {
  return Cat.find(ISSFF, true).list();
}
```

Now that we have the API up and running it's time to test it and see if it performs as we expect.

Let's Talk about Testing

In our experience with working with many developer teams, nothing can dampen the mood of many developers quite like a discussion on the subject of testing that their software does what it's supposed to do.

For many developers, testing is the equivalent of getting a dental checkup—few like doing it but all of us need to do it a lot more. It's a set of bridges that have to be crossed (often under duress) before our precious, handcrafted, artisan-designed piece of software excellence is accepted into the nirvana that is production. Testers are seen as the other team that ensures that we've dotted our I's and crossed our T's, and they don't appreciate how hard we've suffered for our art. Basically, we write it, *throw it over the fence to test*, and they check it.

If you're reading the preceding paragraph and mentally going *yep, yep, that's us, that's us, that's how we roll*, we've got really bad news for you. *You're basically doing it wrong*. It may have made sense when big bang software releases happened every 6-12 months, but in more agile organizations with faster, more frequent releases into production, this approach is considered cumbersome and archaic. There are always exceptions to this, for example, critical control systems, highly regulated environments, and so on, but for the majority of enterprise developers, this isn't the case.

The quality of software is the responsibility of the delivery team, from the Product Owner writing user stories to the engineers writing code and the associated tests. As a wise delivery manager once said, "testing is an activity, not a role." In effective software delivery teams, testing is a continuous activity that spans the entire software development life cycle. There are a number of principles that we try to adhere to when it comes to testing:

1. Automate as much as possible, but not so much that there's no human oversight. There's always value in having users interact with the application under test, in particular when it comes to end-to-end, acceptance, and exploratory testing.
2. Testing code is as important as production code—both need to be kept up to date and removed/deprecated when not adding value.
3. One meaningful test can be worth more than hundreds of scripted test cases.

In *Chapter 7*, *Open Technical Practices – The Midpoint*, we introduced the idea of the Automation Test Pyramid. For each of the different types of tests defined in the pyramid, there are several testing tools and frameworks we use across our PetBattle application.

Generally speaking, we have chosen to use what are considered the default test tools for each of the application technology stacks as these are the simplest to use, are the best supported, have good user documentation, and are generally easy to adopt if people are new to them:

<table>
<tr><th>Application</th><th>Technology</th><th>Unit Test</th><th>Service Test</th><th>E2E Test</th></tr>
<tr><td>API</td><td>Java/Quarkus</td><td>JUnit 5</td><td>REST Assured</td><td rowspan="3">Protractor and Selenium Grid</td></tr>
<tr><td>Tournament</td><td>Java/Quarkus</td><td>JUnit 5</td><td>REST Assured
Test Containers</td></tr>
<tr><td>UI</td><td>Angular</td><td>Jest</td><td>Jest</td></tr>
</table>

Table 15.1: Test Frameworks in use

Let's take a look at some of these tests in more detail.

Unit Testing with JUnit

In both the API and Tournament applications, we have different examples of standard unit tests. Quarkus testing[13] has great support for the standard unit test framework JUnit.[14] The anatomy of all unit tests using this framework is very similar. Let's take a look at the API application `CatResourceTest.java`[15] as an example:

```
@QuarkusTest
class CatResourceTest {

  private static final Logger LOGGER = LoggerFactory
                                    .getLogger("CatResourceTest");

  @Test
  void testCat() {
    PanacheMock.mock(Cat.class);
    Mockito.when(Cat.count())
      .thenReturn(Uni.createFrom().item(23l));
    Assertions.assertEquals(23, Cat.count().await().indefinitely());
```

13 https://quarkus.io/guides/getting-started-testing

14 https://junit.org/junit5/

15 https://github.com/petbattle/pet-battle-api/blob/master/src/test/java/app/battle/CatResourceTest.java

In Java, we use annotations to make our Java class objects (POJOs) into tests. We use the `@QuarkusTest` annotation to bring in the JUnit framework for this class and we can think of the class as a test suite that contains lots of individual tests. Each method is a single test that is annotated with `@Test`. For this unit test, we don't have a database running, so we use mocks[16] for the `Cat.class`. A mock is a fake object. It does not connect to a real database, and we can use it to test the behavior of the `Cat` class. In this case, we are asserting in our test that when we call the method `Cat.count()`, which corresponds to the number of likes of our pet image in PetBattle, we receive back the expected number (23). We use the `Uni` and `await()` functions because we are using the reactive programming model in our Quarkus application.[17]

We run these unit tests as part of the automated continuous deployment pipeline and visualize and report on the tests' success and history using our CI/CD tools, including Jenkins, Tekton, and a test report tool such as Allure.[18]

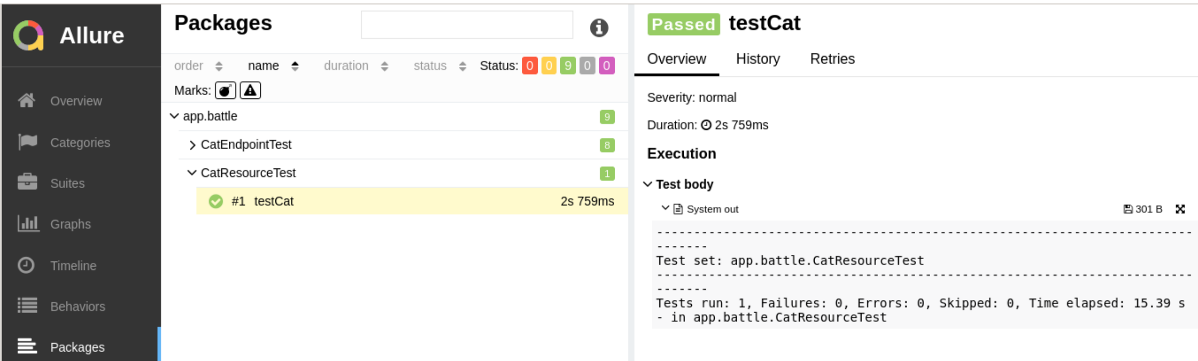


Figure 15.6: Visualization of tests using Allure

In the next section, we'll continue with service and component testing with REST Assured and Jest.

Service and Component Testing with REST Assured and Jest

Jest and REST Assured are **Behavior-Driven Development** (**BDD**) frameworks for JavaScript and Java. We covered BDD in *Chapter 7*, *Open Technical Practices – The Midpoint*. These frameworks make it super easy for developers to write tests where the syntax is obvious and easy to follow.

16 https://quarkus.io/guides/mongodb-panache

17 https://quarkus.io/guides/getting-started-reactive#mutiny

18 https://github.com/allure-framework

We are going to cover the basics of component testing the PetBattle user interface[19] using Jest. The user interface is made of several components. The first one you see when landing on the application is the home page. For the home page component, the test class[20] is called `home.component.spec.ts`:

```
describe('HomeComponent', () => {
  let component: HomeComponent;
  let fixture: ComponentFixture<HomeComponent>;

  beforeEach(async () => {...
  });

  beforeEach(() => {...
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });
});
```

Each test has a similar anatomy:

- `describe()`: The name of the test suite and test specification argument
- `beforeEach()`: Runs the function passed as an argument before running each test
- `it()`: Defines a single test with a required expectation and a function with logic and assertions
- `expect()`: Creates an expectation for the test result, normally with a matching function such as `toEqual()`

So in this case, the unit test will expect the `HomeComponent` to be created correctly when the test is run.

19 https://angular.io/guide/testing

20 https://github.com/petbattle/pet-battle/blob/master/src/app/home/home.component.spec.ts

Similarly, within the API application, REST Assured is a testing tool that allows us to write tests using the familiar `Given, When, Then` syntax from *Chapter 7, Open Technical Practices – The Midpoint*. Let's examine one of the service API tests in the test suite `CatResourceTest.java`[21]:

```
@Test
@Story("Test pet create")
void testCatCreate() {
    CatInstance catInstance = new CatInstance();

    RestAssured.given()
            .contentType(ContentType.JSON)
            .body(catInstance.cat)
            .log().all()
            .when().post("/cats")
            .then()
            .log().all()
            .statusCode(201)
            .body(is(notNullValue()));
}
```

In this test, we are creating a `Cat` object. The `Cat` class is the data object in PetBattle that contains the pet's uploaded image, along with its PetBattle vote count, and is stored in MongoDB. In the test, given the `Cat` object, we use an `HTTP POST` to the `/cats` endpoint and expect a return status code of (`201`), which is `CREATED`. We also test the HTTP response body is not empty. It should contain the ID of the newly created `Cat`:

```
@QuarkusTest
@QuarkusTestResource(MongoTestResource.class)
@Epic("PetBattle")
@Feature("PetEndpointTest")
class CatEndpointTest {
```

21 https://github.com/petbattle/pet-battle-api/blob/master/src/test/java/app/battle/CatResourceTest.java

In this service test, we make use of the `@QuarkusTestResource` annotation to create and start an embedded MongoDB for testing against. So, this test is a bit more sophisticated than the basic unit test that was using mocks only. We also track the execution of these service tests using our test report tool:

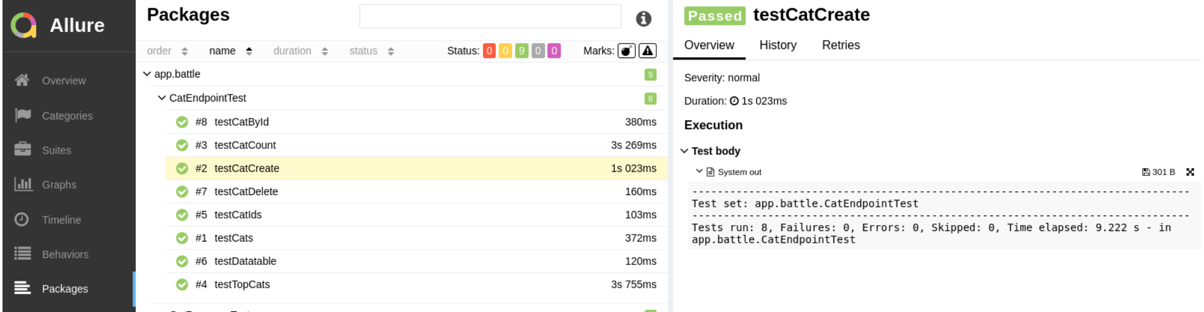


Figure 15.7: Visualization of service tests using Allure

Now we have seen what unit tests look like, let's move up the test pyramid to have a look at service-level testing.

Service Testing with Testcontainers

Integration testing is always substantially harder than unit testing as more components have to be either stood up or simulated/mocked. The next level of testing in our test pyramid is integration testing using a Java framework called **Testcontainers**.[22] Testcontainers allows us to easily create and start components such as MongoDB, **KeyCloak**, and **Infinispan** and perform tests using those components. The following classes instantiate and manage the containers and inject them into the testing life cycle of the Quarkus framework:

```
$ ls src/test/java/com/petbattle/containers/
InfinispanTestContainer.java KeycloakTestContainer.java   MongoTestContainer.
java
```

Within the integration test code at `ITPetBattleAPITest.java`, we just inject the previously created containers and use them as resources during the test:

```
$ head src/test/java/com/petbattle/integration/ITPetBattleAPITest.java

package com.petbattle.integration;
...
```

22 https://www.testcontainers.org/

```
@QuarkusTest
@DisplayName("API Test Cases")
@QuarkusTestResource(MongoTestContainer.class)
@QuarkusTestResource(InfinispanTestContainer.class)
@QuarkusTestResource(KeycloakTestContainer.class)
public class ITPetBattleAPITest {
```

This is a great example of how containers can be used as part of a testing phase. The containers are spun up, the tests are run, and the containers are removed. The only real prerequisite is that the Docker daemon is run on the machine running the tests. To run the integration tests use the command `mvn clean verify -Pintegration`.

End-to-End Testing

Our application is made up of a frontend written in Angular, which makes calls for data to two APIs. One is for tournaments and the other is for cats. We can think of the interplay between these components as the system as a whole. Any time a change is made to either of these individual applications, it should require revalidating the whole system. The end-to-end automated testing is performed primarily in the user interface but exercises the underlying services layer.

There are loads of tools to do testing from the user interface level. Some of the more popular ones are things like Selenium and Cypress, which are used to drive a web application and simulate user behavior. There are pros and cons to each – Selenium is just browser automation so you need to bring your own test frameworks, whereas Cypress is an all-in-one testing framework. Selenium Grid, when running on Kubernetes, allows us to test against multiple browsers in parallel by dynamically provisioning the browser on each test execution, meaning we don't have browsers waiting idly for us to use them.

For our end-to-end testing, we're using Protractor from the Angular team. We already deployed an instance of Selenium Grid built for Kubernetes by the Zalando team (called Zalenium https://opensource.zalando.com/zalenium/) when we deployed our tooling. Zalenium is pretty handy as it allows us to play back previous test runs and watch them live. In your cluster, if you get the route for Zalenium (`oc get routes -n labs-ci-cd`) and append `/grid/admin/live`, you can follow the tests as they execute or go to **/dashboard** to watch the historical test executions.

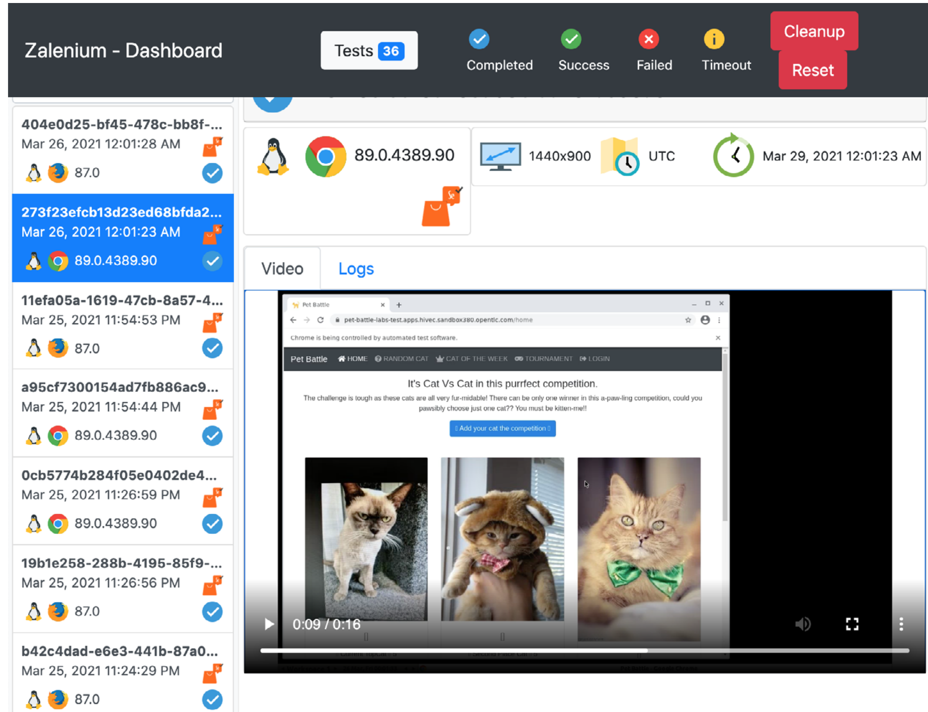


Figure 15.8: Zalenium dashboard showing test history and video playback

Our `system-tests` project (https://github.com/petbattle/system-tests) has all of the system tests that we should execute after any change is pushed to the frontend or backend services. The tests are written using Cucumber-style BDD. In fact, we should be able to connect the BDD to the acceptance criteria from our PetBattle Sprint items.

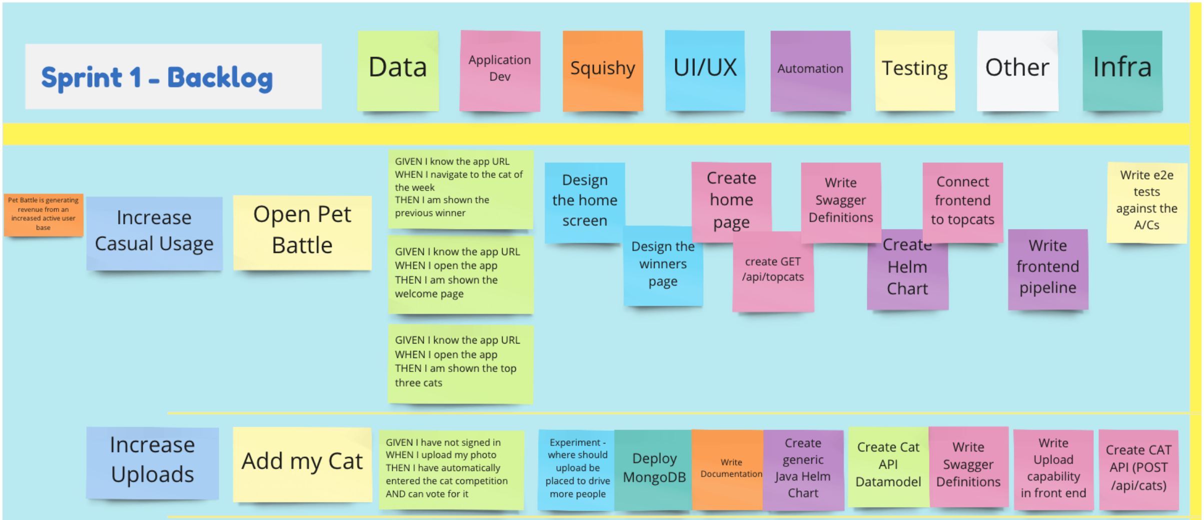


Figure 15.9: Example of BDD written as acceptance criteria on a Sprint board for PetBattle

Here's a test for a tournament feature written in the `Given`, `When`, `Then` syntax:

```
Feature: Tournament features

  Scenario: Should only be prompted to login on navigating to the
            tournament
    Given I am on the home page
    When  I move to the tournament page
    Then  I should be redirected to keycloak
```

The `system-test` project has its own Jenkinsfile, so it's already connected to Jenkins via our seed job. We won't go through the contents of this Jenkinsfile in detail. Suffice to say, the pipeline has two stages, as per our Big Picture, one to run the tests and the other to promote the app if the tests have passed. Explore the code for this in the accompanying Git repo https://github.com/petbattle/system-tests. To extend our Jenkinsfile for `pet-battle` to trigger our system test job, we just need to add another stage to trigger the job. We could use the Jenkins `post{}` block, but we only want to trigger the system tests if we're on `master` or `main` and producing a release candidate.

```
stage("🔨 Trigger System Tests") {
  options {
    skipDefaultCheckout(true)
  }
  agent { label "master" }
  when {
    expression { GIT_BRANCH.startsWith("master") || GIT_BRANCH.startsWith("main") }
  }
  steps {
      echo "TODO - Run tests"
      build job: "system-tests/${SYSTEM_TEST_BRANCH}",
            parameters: [[$class: 'StringParameterValue', name: 'APP_NAME', value: "${APP_NAME}" ],
                          [$class: 'StringParameterValue', name: 'CHART_VERSION', value: "${CHART_VERSION}"]
                          [$class: 'StringParameterValue', name: 'VERSION', value: "${VERSION}"]],
            wait: false
  }
}
```

Figure 15.10: The trigger for connecting our pipelines in Jenkins

There are a few parameters that are passed between the jobs:

- `APP_NAME`: Passed to the job so if the tests are successful, the promote stage knows what app to deploy.
- `CHART_VERSION & VERSION`: Any update to the chart or app needs to be patched in Git so this information is passed by the job that triggers the system tests.

We can run the system tests job manually by supplying this information to the job, but each service with a Jenkinsfile should be able to pass these to the system tests. This job can also be triggered from Tekton too if we were to mix the approach to the pipelines. With the two pipelines wired together, we can trigger one if the webhook is set up by running the following command:

```
$ git commit --allow-empty -m "🍌 kickoff jenkins 🦤" && git push
```

If we now check in the Jenkins Blue Ocean Web UI, we should see the following:

Figure 15.11: The system tests pipeline

On Jenkins, we should see the system tests pipeline running and promoting if successful. The Cucumber reports are also included for the job.

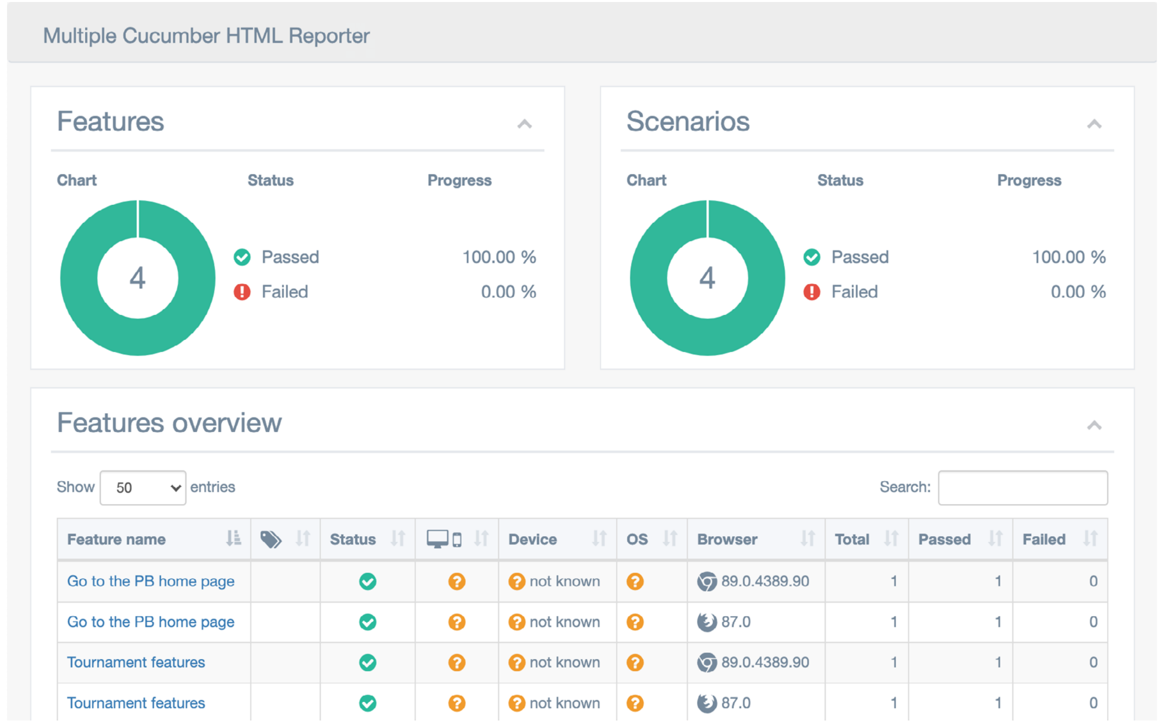


Figure 15.12: The Cucumber report in Jenkins

These provide insight into which cases were executed for what browser and report any failures that may have occurred. Let's switch gear a little now and take a look at non-functional testing.

Pipelines and Quality Gates (Non-functionals)

Quality is often just focused on whether tests pass or not. However there's also the concept of code quality. The code may perform as expected but the manner in which it's been written could be so poor that it could be a future source of problems when changes are added. So now it's time to check the quality of our code.

SonarQube

As part of the Ubiquitous Journey, we have automated the Helm chart deployment of SonarQube, which we are using to test and measure code quality. In `values-tooling.yaml`, the SonarQube stanza references the Helm chart and any extra plugins that are required. Many of the common language profile plugins are already deployed with the base version of SonarQube, for example, Java, JavaScript, and Typescript. We add in extra plugin entries for Checkstyle, our Java formatting check tool, and a dependency checker for detecting publicly disclosed vulnerabilities contained within project dependencies:

```
  # Sonarqube
  - name: sonarqube
    enabled: true
    source: https://github.com/redhat-cop/helm-charts.git
    source_path: "charts/sonarqube"
    source_ref: "sonarqube-0.0.14"
    sync_policy: *sync_policy_true
    destination: *ci_cd_ns
    values:
      initContainers: true
      plugins:
        install:
          - https://github.com/checkstyle/sonar-checkstyle/releases/
download/8.35/checkstyle-sonar-plugin-8.38.jar
          - https://github.com/dependency-check/dependency-check-sonar-
plugin/releases/download/2.0.7/sonar-dependency-check-plugin-2.0.7.jar
```

With the basic SonarQube pod deployed, there is one more piece of configuration we need to automate – the creation of a code quality gate. The quality gate is the hurdle our code must pass before it is deemed ready to release. This boils down to a set of conditions defined in code that specify particular measurements, for example:

- Do we have new blocking issues with the code that was just added?
- Is the code test coverage higher than a given percentage?
- Are there any identifiable code vulnerabilities?

SonarQube lets us define these quality gates[23] using its REST API. For PetBattle, we use a Kubernetes job to define our quality gate `AppDefault` and package it as a Helm chart for deployment. The chart is deployed using Ubiquitous Journey and ArgoCD.

Conditions

Conditions on New Code

Metric	Operator	Value
Coverage	is less than	80.0%
Duplicated Lines (%)	is greater than	3.0%
Maintainability Rating	is worse than	A
Reliability Rating	is worse than	A
Security Rating	is worse than	A

Figure 15.13: A SonarQube quality gate definition

The SonarQube server can be queried via a REST API, whether a recent report against a particular project has passed or failed this quality gate. We have configured a Tekton step and task in our pipelines to automatically check this each time we run a build.

Our PetBattle Java applications are configured using Maven to talk to our SonarQube server pod and generate the SonarQube formatted reports during each build, bake, and deploy. In the reusable `maven-pipeline.yaml`, we call the following target to generate these reports:

```
# code analysis step maven pipeline

    - name: code-analysis
      taskRef:
        name: maven
      params:
        - name: MAVEN_MIRROR_URL
          value: "$(params.MAVEN_MIRROR_URL)"
        - name: MAVEN_OPTS
          value: "$(params.MAVEN_OPTS)"
        - name: WORK_DIRECTORY
          value: "$(params.APPLICATION_NAME)/$(params.GIT_BRANCH)"
        - name: GOALS
          value:
            - install
            - org.owasp:dependency-check-maven:check
```

23 https://docs.sonarqube.org/latest/user-guide/quality-gates/

```
              - sonar:sonar
          - name: MAVEN_BUILD_OPTS
            value:
              - '-Dsonar.host.url=http://sonarqube-sonarqube:9000'
              - '-Dsonar.userHome=/tmp/sonar'

  # code analysis step nodejs pipeline

      - name: code-analysis
        taskRef:
          name: nodejs
        params:
          - name: NPM_MIRROR_URL
            value: "$(params.NPM_MIRROR_URL)"
          - name: GOALS
            value:
              - "run"
              - "sonar"
```

Similarly, for the PetBattle UI using `nodejs`, we can configure the client to call SonarQube as part of its Tekton pipeline. Once these steps have successfully run, we can explore the SonarQube Web UI and drill down into any areas to find out more information.

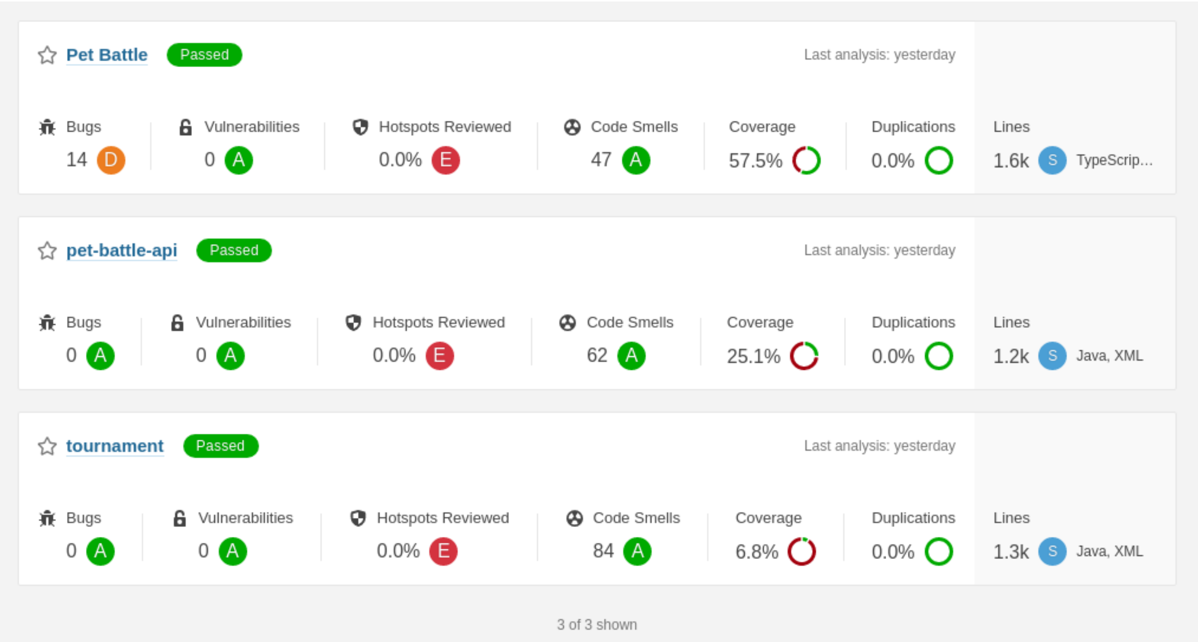


Figure 15.14: SonarQube project view

In a bit of recent development for A/B testing support in the PetBattle UI, some code bugs seemed to have crept in! Developers can drill down and see exactly what the issues are and remediate them in the code base. SonarQube ranks issues based on severity defined in the Language Quality Profile, which can be altered to suit your development code quality needs.

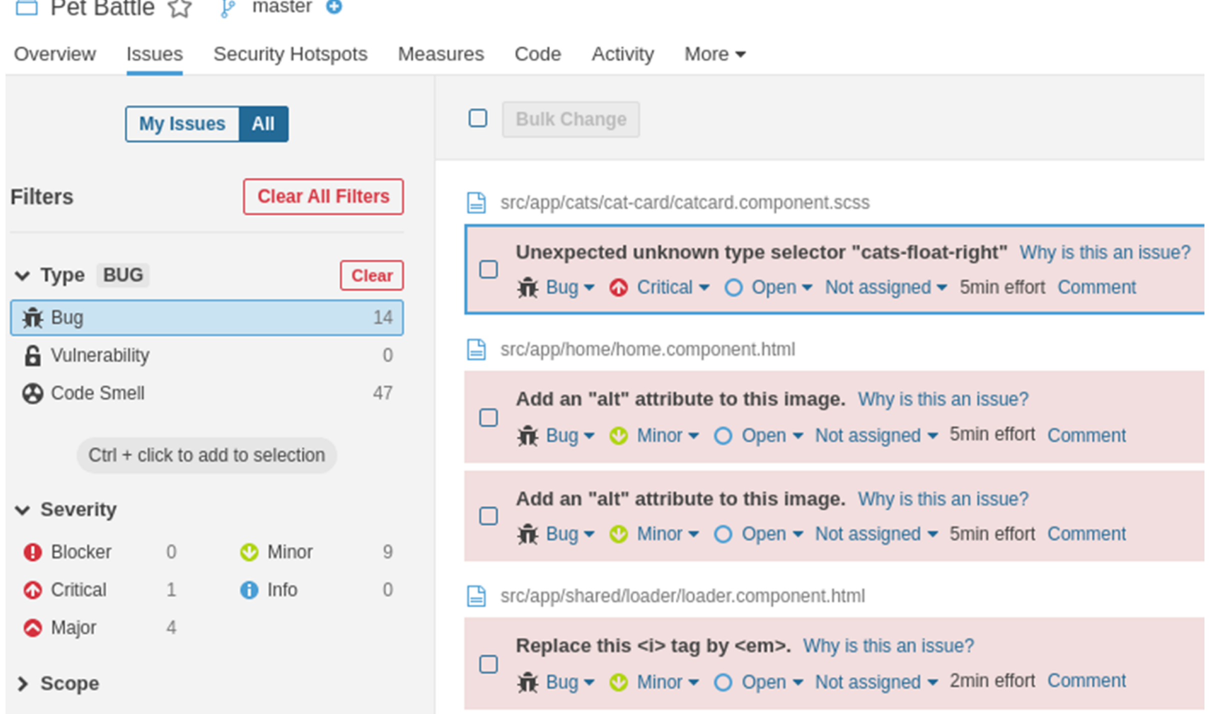


Figure 15.15: SonarQube drilling into some bug details

SonarQube also reports on the last run's code testing coverage. On the code base side, you generate coverage reports using the `LCOV`[24] format, so in Java, this is done by `JaCoCo`[25] and in JavaScript, the coverage reports are produced by the `mocha/jasmine` modules. These reports are uploaded into SonarQube and give the team visibility into which parts of their code base need more testing. A nice way to view this information is using the heatmap, which visualizes the bits of code that have near 100% coverage (green), down to areas that are not covered at all 0% (red). The statistics are also reported – the percentage coverage overall, the number of lines covered, and so on.

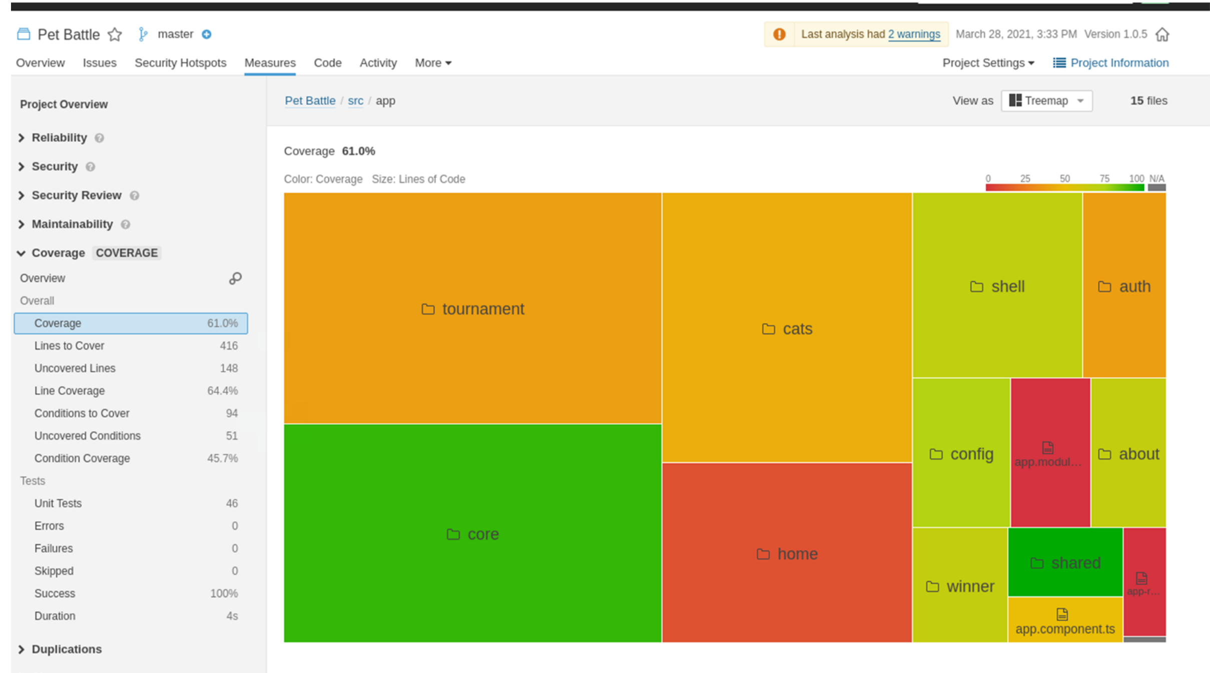


Figure 15.16: SonarQube test coverage heatmap for PetBattle

24 https://github.com/linux-test-project/lcov

25 https://www.eclemma.org/jacoco/

The last plugin we use for our Java applications is the OWASP Dependency-Check plugin.[26] We move security checking "left" in our pipeline. In other words, we want to discover early in the development process when security vulnerabilities or CVEs are creeping into our applications' dependencies. By identifying which dependencies are vulnerable to a CVE early as part of the build cycle, developers are in a much better position to update them, rather than finding there are issues once our applications are deployed.

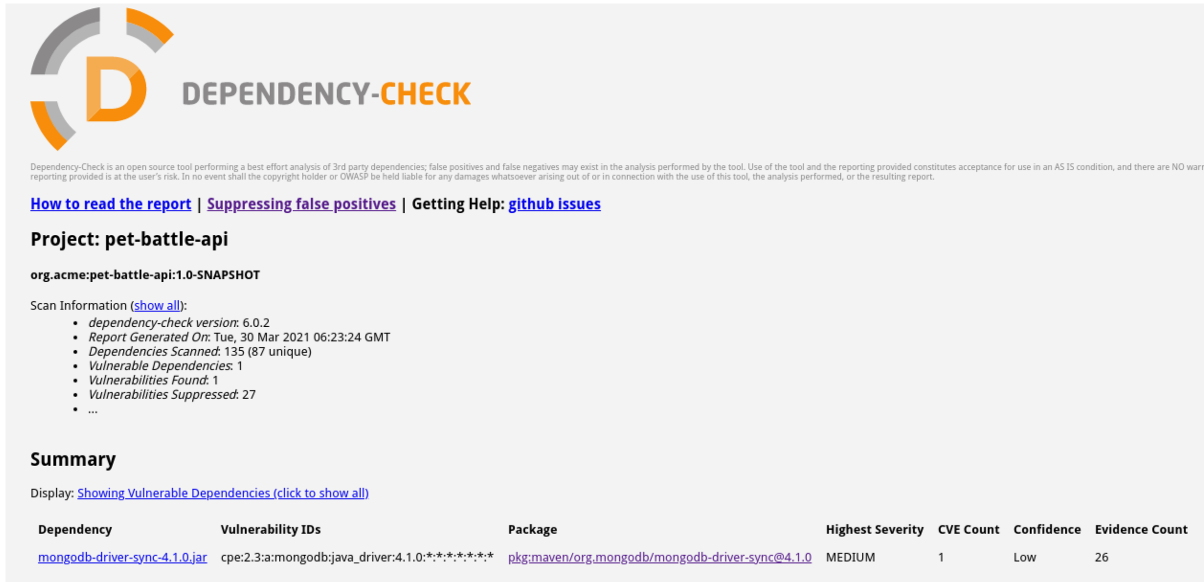


Figure 15.17: Dependency-Check plugin report

The plugin sources data from multiple open source resources including the US National Vulnerability Database[27] and Sonatype OSS Index.[28] In conjunction with security team members, developers can verify known vulnerabilities and suppress any false positives using a configuration file. The report is very detailed and includes links to these sites to assist CVE identification and reporting.

26 https://github.com/dependency-check/dependency-check-sonar-plugin

27 https://nvd.nist.gov/

28 https://ossindex.sonatype.org/

Perf Testing (Non-Functional)

One of our favorite command-line tools for getting fast feedback on the performance of REST APIs is a tool called **hey**.[29] There are a lot of similar tools available. Apache Bench[30] is probably the most venerable.

```
@4fc082b42236:~
virt:~$ hey -t 30 -c 10 -n 100 -H "Content-Type: application/json" -m GET
 https://pet-battle-api-labs-test.apps.hivec.sandbox1405.opentlc.com/cats

Summary:
  Total:        4.2048 secs
  Slowest:      0.6216 secs
  Fastest:      0.3155 secs
  Average:      0.4115 secs
  Requests/sec: 23.7823


Response time histogram:
  0.316 [1]     |■
  0.346 [2]     |■
  0.377 [12]    |■■■■■■■■
  0.407 [58]    |■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
  0.438 [17]    |■■■■■■■■■■■■
  0.469 [0]     |
  0.499 [0]     |
  0.530 [2]     |■
  0.560 [2]     |■
  0.591 [2]     |■
  0.622 [4]     |■■■


Latency distribution:
  10% in 0.3729 secs
  25% in 0.3888 secs
  50% in 0.3992 secs
  75% in 0.4122 secs
  90% in 0.5086 secs
  95% in 0.5713 secs
  99% in 0.6216 secs

Details (average, fastest, slowest):
  DNS+dialup:   0.0081 secs, 0.3155 secs, 0.6216 secs
  DNS-lookup:   0.0003 secs, 0.0000 secs, 0.0037 secs
  req write:    0.0000 secs, 0.0000 secs, 0.0002 secs
  resp wait:    0.0631 secs, 0.0365 secs, 0.0906 secs
  resp read:    0.3402 secs, 0.2477 secs, 0.4536 secs

Status code distribution:
  [200] 100 responses
```

Figure 15.18: A simple hey run

29 https://github.com/rakyll/hey

30 https://httpd.apache.org/docs/2.4/programs/ab.html

We like **hey** because it is small, fast, written in Golang, and reports statistics in a format we can easily understand. In the preceding screenshot, we can see a very simple invocation using `hey` on the command line to call the PetBattle API and list all of the pets. We pass in some parameters that represent:

- `-c`: Number of workers to run concurrently
- `-n`: Number of requests to run
- `-t`: Timeout for each request in seconds

We can see the summary statistics reported, and this is the bit we love – a histogram of latency distribution, HTTP status code distribution, as well as DNS timing details. This is super rich information. Histograms are graphs that display the distribution of the continuous response latency data. A histogram reveals properties about the response times that the summary statistics cannot. In statistics, summary data is used to describe the complete dataset – minimum, maximum, mean, and average, for example. **Hey** gives us these summary statistics at the top of the output.

The graph brings the data to life as we can start to understand the distribution of the latency response over the time the test ran. Over the 4.2 seconds it took to send the 100 requests, we can see that most of the data is clustered around the 0.4-second mark, which is nearly 50% of all traffic. Often, in service performance design, we are interested in what the 95% or 99% percentile number is. That is, for all of the sample data, what the response latency is for 95% (or 99%) of the traffic. In this test run, it is measured at 0.57 seconds – in other words, 95% of the data was at or below this mark.

The shape of the histogram is also important. Where are the response latencies grouped? We can easily see if the response times are distributed evenly around the mean (Gaussian) or if they have a longer or shorter tail. This can help us characterize the performance of the service under various loads. There are many types of load profiles you could use, for example, burst loads where we throw a lot of instantaneous traffic at our API, compared to more long-lived soak tests under a lower load. You might even have known loads from similar applications in production already. A great open source tool for designing these types of test loads, which can model threading and ramping really well, is Apache JMeter[31] and we highly recommend it as a tool to have in your toolbox. To keep things simple, we won't cover that tool here.

31 https://jmeter.apache.org/

The two diagrams shown in *Figure 15.19* display simple load tests. The one on the left is a burst type of test – 300 consecutive users calling 900 times to our PetBattle API. We can see the 95% is 15.6 seconds – this is quite a long time for users to wait for their cats! The one on the right is a soak test – 50 consecutive users calling 10,000 times to our PetBattle API. A very different set of statistics: a test duration of 461 seconds, and the 95% is 2.8 sec—much better from an end user's perspective.

At this point, it is important to think about what the test is actually doing and how it relates to the PetBattle application suite in general. If we think about it, the test may not be totally indicative of the current user interface behavior. For example, we do not perform a call to return all of the images in our MongoDB at once but rather page the results. And there are of course other API endpoints to test, for example, the `topcats` API, which returns the top three most popular pets and is called every time you visit the home page. We are returning the test dataset we have loaded into PetBattle, that is, around 15 pet images, so it is not a massive amount of data. It's important to always step back and understand this wider context when we run performance tests so we don't end up testing the wrong thing!

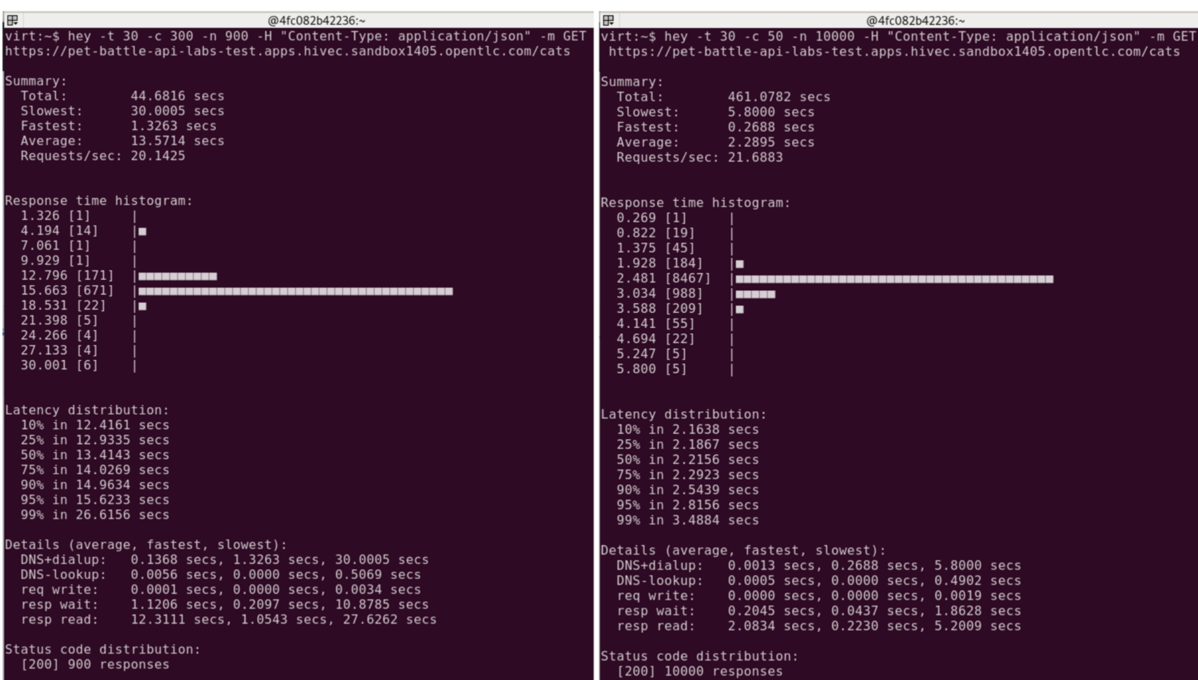


Figure 15.19: Burst and soak tests against the PetBattle API

Nonetheless, this is good data to ponder. A good result is that both the soak and burst tests only returned HTTP 200 response statuses – there were no error responses from the API. That gives us confidence that we have not broken anything or reached any internal system limits yet. We can also examine the details to make sure DNS resolution is not causing issues from the client-calling perspective.

Now we are familiar with the client or calling side of performance testing, let's switch to the PetBattle API application running on the server side. If we browse to the **Developer** view and select the `pet-battle-api` pod in the `labs-test` namespace, we can see some important server-side information:

- The PetBattle API is autoscaled to two pods.
- Monitoring metrics for the pods (check the appendix if you haven't enabled this for CRC).

As a developer, we have configured the PetBattle API application to use the **Horizontal Pod Autoscaler** (**HPA**). This specifies how the OpenShift Container Platform can automatically increase or decrease the scale of a replication controller or deployment configuration, the number of running pods, based on the metrics collected from the pods that belong to our application.

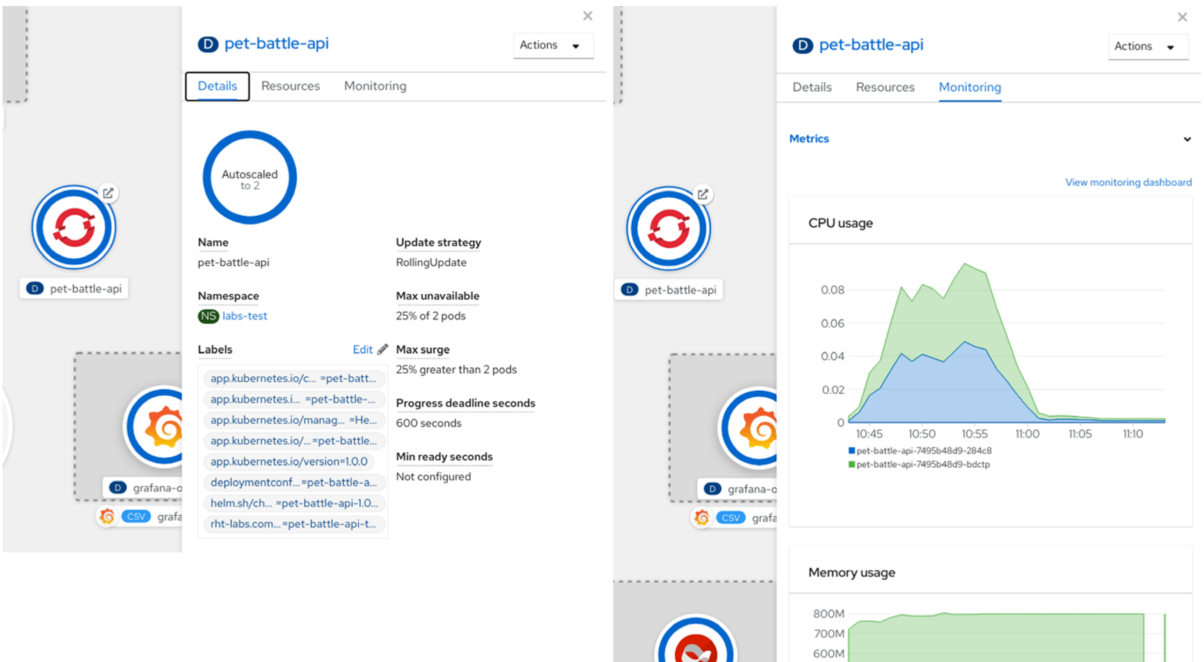


Figure 15.20: PetBattle API pod in the labs-test namespace

In our PetBattle API Helm chart, we specified the HPA with configurable values for minimum pods, maximum pods, as well as the average CPU and memory targets. Using hey, we can now test out various scenarios to help us tune the PetBattle API application under load:

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: {{ include "pet-battle-api.fullname" . }}
  labels:
    {{- include "pet-battle-api.labels" . | nindent 4 }}
spec:
  scaleTargetRef:
    {{- if .Values.deploymentConfig }}
    apiVersion: v1
    kind: DeploymentConfig
    {{- else }}
    apiVersion: apps/v1
    kind: Deployment
    {{- end }}
    name: {{ include "pet-battle-api.fullname" . }}
  minReplicas: {{ .Values.replicas.min }}
  maxReplicas: {{ .Values.replicas.max }}
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: AverageValue
          averageValue: {{ .Values.hpa.cpuTarget }}
    - type: Resource
      resource:
        name: memory
        target:
          type: AverageValue
          averageValue: {{ .Values.hpa.memTarget }}
```

We initially took a rough guess at these settings in our HPA, for example, `min replicas = 2`, `max replicas =6`, `CPU = 200m`, `mem = 300Mi`, and set the resource limits and requests in our Deployment appropriately. We always have a minimum of two pods, for high availability reasons. The HPA is configured to scale based on the average memory and CPU loads. We don't yet understand whether the application is memory- or CPU-intensive, so choose to scale based on both these measurements.

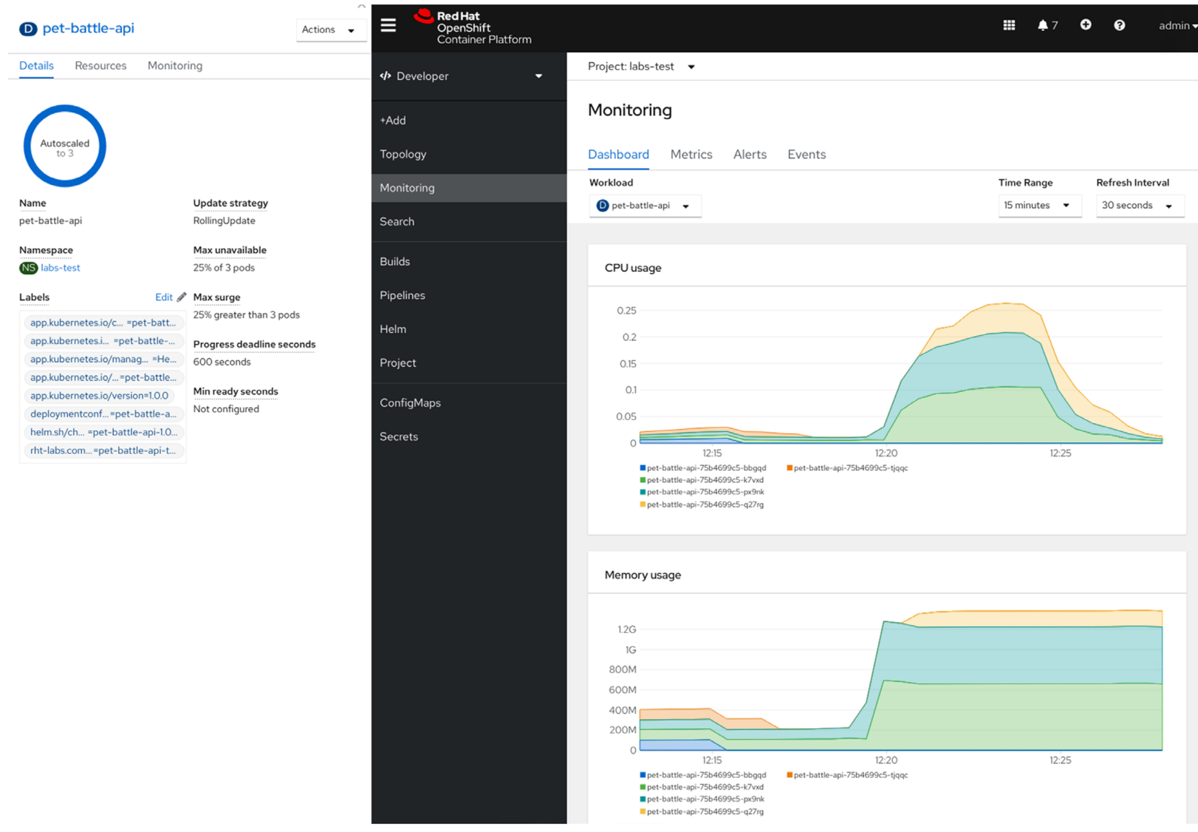


Figure 15.21: PetBattle API HPA in action, scaling pods under load

We use hey to start a burst workload, 400 concurrent requests, and watch the behavior of the HPA as it starts more pods to keep to the specified memory and CPU averages. Once the test concludes, the HPA scales our workload back down to the minimum as the application recovers resources, in this case through Java garbage collection. OpenShift supports custom metrics for the HPA as well as other types of pod scalers, for example, the Vertical Pod Autoscaler.[32]

To conclude this section, we want to point out one more Kubernetes object that the developer needs in their toolbelt – the **Pod Disruption Budget** (**PDB**). Again, using a Helm chart template for the PDB, we can limit the number of concurrent disruptions that the PetBattle API application experiences. By setting up a PDB, we can allow for higher availability while permitting the cluster administrator to manage the life cycle of the cluster nodes.

32 https://docs.openshift.com/container-platform/4.7/nodes/pods/nodes-pods-using.html

If the cluster is being updated and nodes are being restarted, we want a minimum of one `pet-battle-api` pod available at all times:

```
$ oc get pdb

NAME          MIN AVAILABLE  MAX UNAVAILABLE   ALLOWED DISRUPTIONS    AGE
pet-battle-api    1            N/A                     1               46h
```

This ensures a high level of business service for our PetBattle API. We can see `ALLOWED_DISRUPTIONS` is set to 1 – this is because, at the time, the HPA had scaled the number of available replicas to 3 and this will change as the number of available pods changes.

One of the great things about performance testing applications on OpenShift is that all of the tools are at a developer's fingertips to be able to configure, test, measure, and tune their applications to achieve high availability and performance when under load. Each application service is independently scalable, tunable, and deployable, which makes for a faster and targeted feedback loop when dealing with scale and performance issues.

In the next section, we are going to take a look at what makes a good OpenShift Kubernetes citizen, automating Kubernetes resource validation as part of our pipeline.

Resource Validation

One aspect of testing that doesn't yet get much thought is the quality of the Kubernetes resources being deployed on the cluster. For applications to be considered *good citizens* on Kubernetes, there are a number of deployment best practices to be followed—including health checks, resource limits, labels, and so on—and we will go through a number of these in in *Chapter 16*, *Own It*. However, we need to validate the resource definitions being applied to the cluster to ensure a high level of compliance to not only industry recommendations but also any other resource recommendations that we see fit to add. This is where **Open Policy Agent** (**OPA**)[33] and associated tools can come into play. This enables us to validate resource definitions during a CI pipeline and also when applying resources to a cluster. OPA by itself is a policy validator and the policies are written using a language called Rego. Additional OPA tools such as Conftest[34] and Gatekeeper[35] add a lot of value and governance from a usability and deployment perspective. OPA is also embeddable into other third-party tools such as KubeLinter.[36]

33 https://www.openpolicyagent.org/

34 https://github.com/open-policy-agent/conftest

35 https://github.com/open-policy-agent/gatekeeper

36 https://github.com/stackrox/kube-linter

We haven't used OPA's server-side validation component, Gatekeeper,[37] as part of PetBattle but there are example Rego policies in the Red Hat Community of Practice GitHub repo[38] that are definitely worth exploring. If this is something of interest to you, definitely check out the blog on OpenShift.com that details setting up all of these components.[39]

However, to show how easy it is to use client-side resource validation and why you should include at least some resource validation in a pipeline, a simple Rego example has been created. Rego policies are easy enough to write and the Rego playground[40] is a great place to write and verify policies, so check it out.

Let's get into an example. In our Non-Functional Requirements Map, we said we wanted to be consistent with our labeling. It makes sense that we should adopt the Kubernetes best practice that suggests the `app.kubernetes.io/instance` label should be on all resources, so let's see how we can write a test to this effect and add it to our pipeline in Jenkins.

The makeup of a policy that denies the creation of a resource is simple enough. A message is formed and passed back to the interpreter if all of the statements are true in the rule. For example, we have written a policy that checks that all resources conform to Kubernetes best practice for naming conventions. The policy here is checking whether `app.kubernetes.io/instance` exists on the resource supplied to it (`input`). If each statement is true, then a message is returned as the error, guiding someone to fix the issue:

```
deny[msg] {
  label := "app.kubernetes.io/instance"
  not input.metadata.labels[label]
  msg := sprintf("\n%s: does not contain all the expected k8s labels
                in 'metadata.labels'.\n Missing '%s'. \nSee: https://
kubernetes.io/docs/concepts/overview/working-with-objects/common-labels",
[input.kind, label])
}
```

37 https://github.com/open-policy-agent/gatekeeper

38 https://github.com/redhat-cop/rego-policies

39 https://www.openshift.com/blog/automate-your-security-practices-and-policies-on-openshift-with-open-policy-agent

40 https://play.openpolicyagent.org/

We can combine this rule with Conftest and a Helm template to create a way to statically validate our resources. In the PetBattle frontend code, there is a policy folder that has a few more policies to check whether all the standard Kubernetes labels[41] are set on our generated resources after we run the `helm template` command. By running a few commands, we can verify these are in place. First, we template our chart to produce the Kubernetes resources we will apply in deploying our software, and secondly, we tell Conftest to check each file generated against the rule:

```
# from the pet battle front end repository (https://github.com/petbattle/
pet-battle.git)
$ for file in $(ls policy/helm-output/pet-battle/templates/); do conftest
test policy/helm-output/pet-battle/templates/$file; done

6 tests, 6 passed, 0 warnings, 0 failures, 0 exceptions
FAIL - policy/helm-output/pet-battle/templates/deploymentconfig.yaml -
DeploymentConfig: does not contain all the expected k8s labels in 'metadata.
labels'.
 Missing 'app.kubernetes.io/name'.
See: https://kubernetes.io/docs/concepts/overview/working-with-objects/
common-labels

6 tests, 5 passed, 0 warnings, 1 failure,  0 exceptions
6 tests, 6 passed, 0 warnings, 0 failures, 0 exceptions
6 tests, 6 passed, 0 warnings, 0 failures, 0 exceptions
```

When executing the rules from the command line, we get a good insight into what's missing from our chart. Of course, we could just assume that we'd always make our charts adhere to the best practices, but the `jenkins-agent-helm` has also got the Conftest binary so we can execute the preceding statements in our Jenkins pipeline too. This example might seem simple but, hopefully, it gives you some idea of the things that can be automated and tested that might seem less obvious.

41 https://kubernetes.io/docs/concepts/overview/working-with-objects/common-labels/#labels

Image Scanning

Red Hat provides the Quay Container Security Operator in OpenShift to bring Quay and Clair image scanning and vulnerability information into our OpenShift cluster. Any container image that is hosted on Quay.io is scanned by Clair.

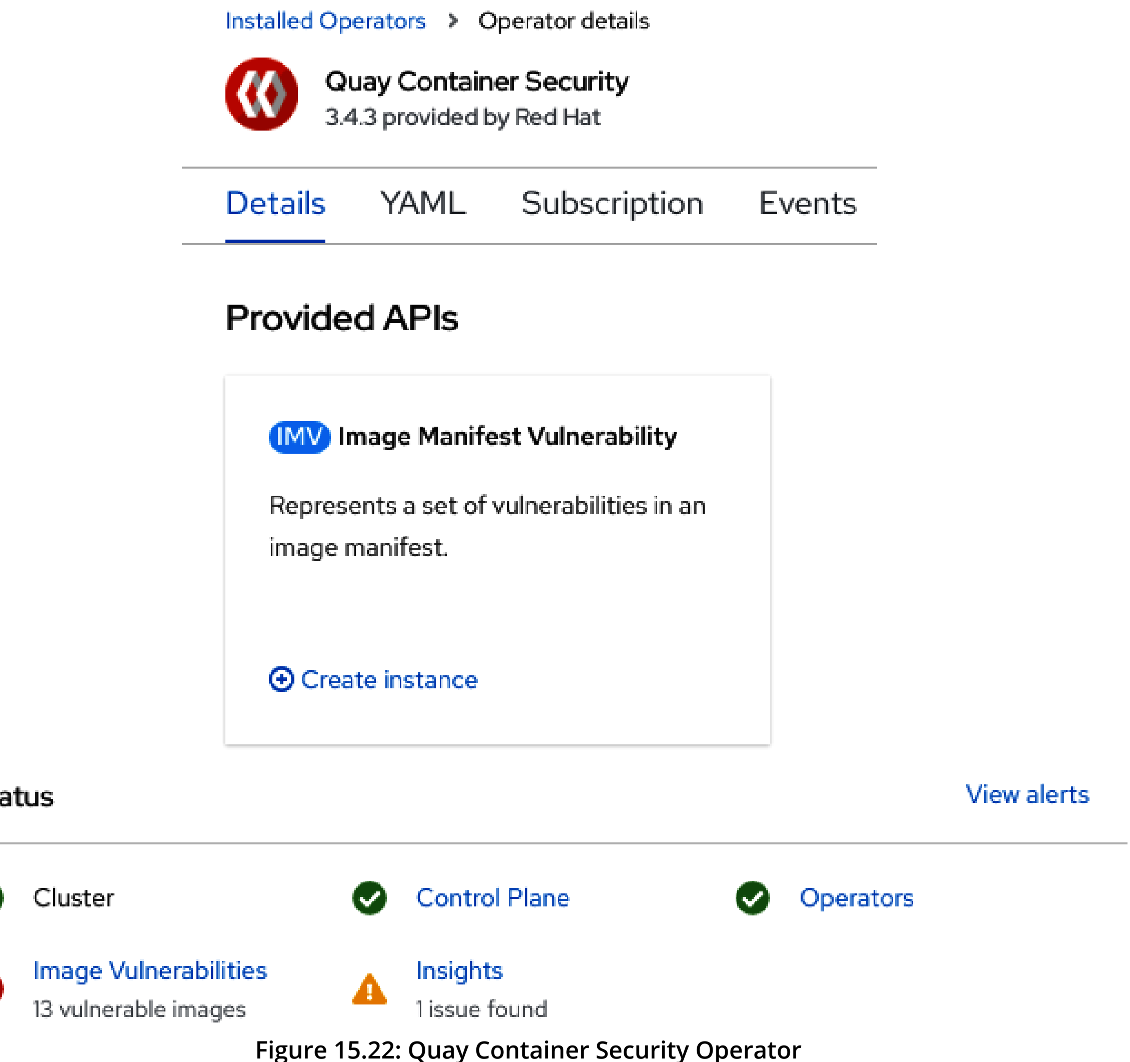


Figure 15.22: Quay Container Security Operator

Any image vulnerability data is exposed back in the OpenShift Web UI so that users and administrators can easily view which images are considered vulnerable and which namespace they are deployed to.

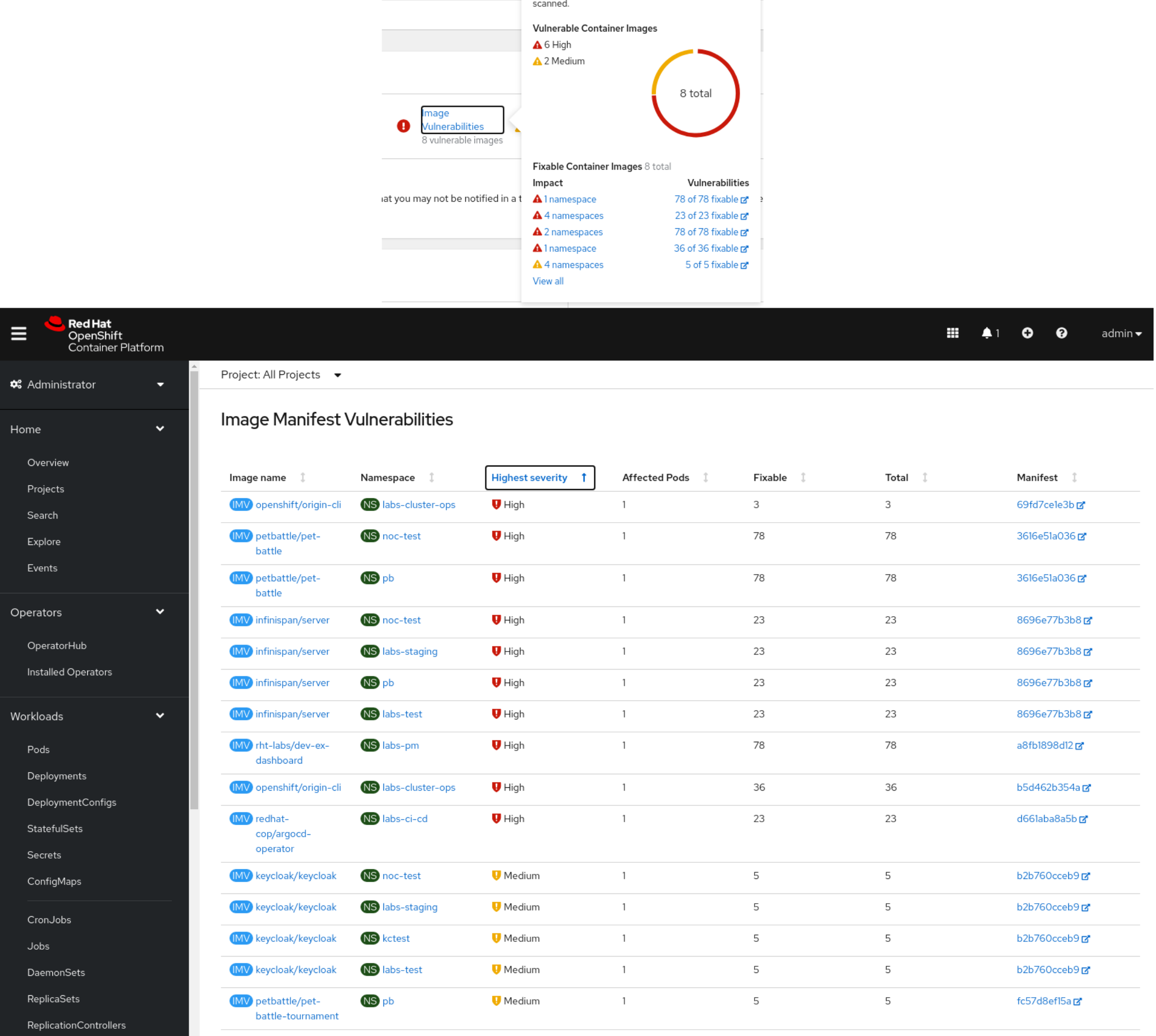


Figure 15.23: Vulnerable container images

With this operator deployed, the OpenShift overview status displays image vulnerability data, which an operator can drill into to find out the status of container images running on the platform. For PetBattle, we don't have any enforcement for image vulnerabilities discovered in our cluster. If we wanted to move the security scanner "left" in our deployment pipeline, there are some great open source scanning tools available on the OpenSCAP website.[42]

Other Non-functional Testing

There are lots of other types of testing we can do to validate our application. In this section are some of the things we feel are important to include in a pipeline, but the reality is there is much more than just this list and books could be written on this topic in and of itself!

Linting

A linter is a static code analysis tool that can check a code base for common pitfalls in design or stylistic errors. This does not check the compiled application, but the structure of the application. This is super important for languages that are not compiled, such as JavaScript. Browsers can interpret JavaScript in different ways so consistency is super critical.

If you think about a large enterprise application, there could be hundreds of developers working on the one code base. These developers could even be globally distributed with different teams looking after different parts of the application's life cycle. Having consistency in the approach to writing the software can dramatically improve maintenance costs. JavaScript is very flexible in how you can write it, whether this is from a functional programming standpoint or object-oriented, so it is important to get this consistency right.

The PetBattle frontend uses TSLint/ESLint[43] to check the style of the code adheres to a standard set of rules. These rules can be manipulated by the team, but the rules are checked into Git so if someone was to disable them or manipulate them, it would be noticed. Our Jenkins pipeline is configured to automatically check the code base using the `npm lint` command and our build will fail if a developer does not adhere to the standard.

---

42 https://www.open-scap.org

43 https://eslint.org/

```
pet-battle — donal@dspring-mac — ..le/pet-battle — -zsh — Solarized Dark ansi — 105×23
#( 04/01/21@11:14pm )( donal@dspring-mac ):~/Documents/Clients/petbattle/pet-battle@master✔
[   npm run lint                                                                              ]

> pet-battle@1.3.1 lint /Users/donal/Documents/clients/petbattle/pet-battle
> ng lint && stylelint "src/**/*.scss" --syntax scss && htmlhint "src" --config .htmlhintrc

TSLint's support is discontinued and we're deprecating its support in Angular CLI.
To opt-in using the community driven ESLint builder, see: https://github.com/angular-eslint/angular-eslin
t#migrating-an-angular-cli-project-from-codelyzer-and-tslint.
Linting "pet-battle"...
All files pass linting.
TSLint's support is discontinued and we're deprecating its support in Angular CLI.
To opt-in using the community driven ESLint builder, see: https://github.com/angular-eslint/angular-eslin
t#migrating-an-angular-cli-project-from-codelyzer-and-tslint.
Linting "pet-battle-e2e"...
All files pass linting.
Browserslist: caniuse-lite is outdated. Please run next command `npm update`

   Config loaded: .htmlhintrc

Scanned 12 files, no errors found (39 ms).
#( 04/01/21@11:20pm )( donal@dspring-mac ):~/Documents/Clients/petbattle/pet-battle@master✔
```

Figure 15.24: Linting PetBattle's frontend locally scans both the JavaScript and HTML

For Java Quarkus apps, Checkstyle[44] is used to analyze the code base.

For Kubernetes resources, the aforementioned Open Policy Agent can assist, and Helm also has the `helm lint`[45] command to validate your charts.

Code Coverage

So, you've written a load of tests and you think things are going great – but how do you know your tests are any good and covering all parts of the code base? Allow me to introduce code coverage metrics! A code coverage reporter is a piece of software that runs alongside your unit test suites to see what lines of code are executed by the tests and how many times. Coverage reports can also highlight when if/else control flows within an application are not being tested. This insight can provide valuable feedback as to areas of a system that remain untested and ultimately reduce the number of bugs.

44 https://checkstyle.sourceforge.io/

45 https://helm.sh/docs/helm/helm_lint/

Our PetBattle frontend is configured to run a coverage report when our Jest tests execute. Jest makes generating the report very simple as it has a flag that can be passed to the test runner to collect the coverage for us. The coverage report is run on every execution of the build and so should be reported through Jenkins.

```
pet-battle — donal@dspring-mac — ..le/pet-battle — -zsh — Solarized Dark ansi — 105×33
  home.component.ts                  |   21.05 |        0 |    9.09 |   14.29 | 21-80
 app/shared                          |     100 |      100 |     100 |     100 |
  index.ts                           |     100 |      100 |     100 |     100 |
  shared.module.ts                   |     100 |      100 |     100 |     100 |
 app/shared/loader                   |     100 |      100 |     100 |     100 |
  loader.component.html              |     100 |      100 |     100 |     100 |
  loader.component.ts                |     100 |      100 |     100 |     100 |
 app/shell                           |     100 |      100 |     100 |     100 |
  shell.component.html               |     100 |      100 |     100 |     100 |
  shell.component.ts                 |     100 |      100 |     100 |     100 |
  shell.service.ts                   |     100 |      100 |     100 |     100 |
 app/shell/header                    |   76.47 |       25 |      60 |   71.43 |
  header.component.html              |     100 |      100 |     100 |     100 |
  header.component.ts                |      75 |       25 |      60 |   69.23 | 25-32
 app/tournament                      |   45.45 |        0 |   16.67 |   41.25 |
  cat404.component.ts                |     100 |      100 |     100 |     100 |
  tournament.component.html          |     100 |      100 |     100 |     100 |
  tournament.component.ts            |   37.74 |        0 |   13.04 |      36 | 34,38-39,43-85,93-111
  tournament.service.ts              |   48.28 |      100 |   16.67 |   42.31 | 21-27,41-43,54-101
 app/winner                          |     100 |      100 |     100 |     100 |
  winner.component.html              |     100 |      100 |     100 |     100 |
  winner.component.ts                |     100 |      100 |     100 |     100 |
 environments                        |     100 |      100 |     100 |     100 |
  .env.ts                            |     100 |      100 |     100 |     100 |
  environment.ts                     |     100 |      100 |     100 |     100 |
-------------------------------------|---------|----------|---------|---------|-----------------------

Test Suites: 19 passed, 19 total
Tests:       46 passed, 46 total
Snapshots:   0 total
Time:        31.624 s
#( 04/01/21@11:14pm )( donal@dspring-mac ):~/Documents/Clients/petbattle/pet-battle@master✔
```

Figure 15.25: Code coverage report from the frontend unit tests locally

When executing our tests in the Jenkins pipeline, we have configured Jest to produce an HTML report that can be reported by Jenkins on the **jobs** page. For any build execution, the report is added to the **jobs** home page. The report will allow us to discover what lines are being missed by our tests. Being able to drill into a report like this can give a good insight into where our testing is lacking.

Back to master index Zip

All files / app/cats/cat-card catcard.component.ts

58.82% Statements 20/34 5.26% Branches 1/19 50% Functions 3/6 56.25% Lines 18/32

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

```
1  3x import { Component, OnInit, Input } from '@angular/core';
2  3x import { CatsService } from '../cats.service';
3  3x import { ConfigurationLoader } from '@app/config/configuration-loader.service';
4  3x import { MatomoTracker } from 'ngx-matomo-v9';
5
6     @Component({
7       selector: 'app-catcard',
8       templateUrl: './catcard.component.html',
9       styleUrls: ['./catcard.component.scss']
10    })
11 3x export class CatcardComponent implements OnInit {
12      public image: string;
13 1x   public currentVoteCount = 0;
14 1x   public spinner = true;
15 1x   public compOver = false;
16      private catId: string;private analytics = false;
17 1x
18      constructor(
19        private catsService: CatsService,
20 1x     private configSvc: ConfigurationLoader,
21 1x     private matomoTracker: MatomoTracker
22 1x   ) {}
23
24      getNewCat(vote: boolean) {
25        if (this.analytics) {
26 1x       // instrument the buttons
27 1x       const voting = vote ? 'UP_VOTE' : 'DOWN_VOTE';
28 1x       this.matomoTracker.trackEvent('A/B Tests', 'Voting', voting);
29 1x     }
30 1x     this.spinner = true;
31 1x     const body = { image: this.image, id: this.catId, count: this.currentVoteCount, vote };
32        vote ? this.currentVoteCount++ : this.currentVoteCount--;
```

Figure 15.26: Code coverage report in Jenkins gives us detailed insight

So, what should I do with these results? Historically, we have worked where coverage is low. It can serve as a great talking point to bring up in a Retrospective. Printing out the reports and discussing them as a team is a great way to assess why the team is struggling to write enough tests. Sometimes teams are drowning by being overwhelmed with pressure to churn out features and so testing can slip to the wayside. Having a coverage reporter in your build can help keep a team honest. You could even set thresholds so that if testing coverage falls below a certain percentage (some teams aim for 80% and above), the build will fail, thus blocking the pipeline until the quality is increased.

Untested Software Watermark

I worked on a project a long time ago that was poorly structured. I was a member of the DevOps team, which I know now is an antipattern in most implementations! This project had many issues, the team had planned three Sprints in advance but hadn't allocated enough time for testing. It was always the thing that got squeezed.

Through Retrospectives with the teams, we discovered that there was simply not enough time for tests. This may sound hard to hear, but the root cause for this was not laziness by the team or a lack of skills; it really was time. The project was running in 8-week blocks that were pre-planned from the beginning with a fixed output at the end. The team thought they were doing Scrum, but in actual fact, they had milestones of functionality to accomplish each sprint and there was no feedback loop. Of course, none of the Scrum team members were involved in the sizing or planning ceremonies either. This meant the teams were constantly under pressure to deliver.

Through a Retrospective, we decided to try to radiate some of this pressure the teams were under as we were not happy that quality was being sacrificed for some arbitrary deadlines. Knowing that failing the pipeline simply would not work for these customers, we had to get creative in showing the software quality. We decided to inject a watermark into any application that had low test coverage. This watermark resembled a DRAFT logo you would find on any document, but ours was a little different.

A large banner reading UNTESTED SOFTWARE was placed across the applications that failed the tests. This watermark did not affect the user behavior of the app; it was just an overlay but it was an amazing way to get people talking. Seeing a giant banner saying UNTESTED is a surefire way to have people question why things have gotten this way.

Let's look at some other ways we can visualize risks during continuous delivery.

The OWASP Zed Attack Proxy (ZAP)

Security scanning is always a hot topic. From image scanning, which we discussed earlier, to dependency checking for our application that happens in our pipelines, there are limitless numbers of things to automate from a security perspective. Let's take another example of something that can be useful to include in a pipeline – the OWASP Zed Attack Proxy.[46]

From their website: *The OWASP Zed Attack Proxy (ZAP) is one of the world's most popular free security tools which lets you automatically find security vulnerabilities in your applications. This allows the developers to automate penetration testing and security regression testing of the application in the CI/CD pipeline.*

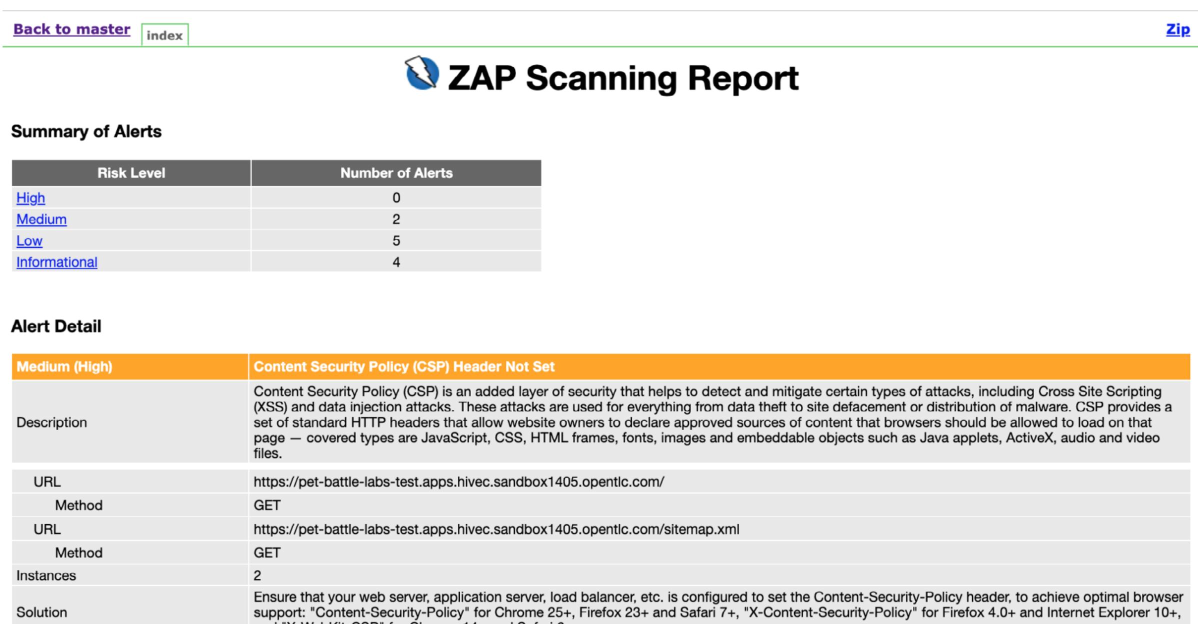
Adding the ZAP security scanning tool to our pipelines is simple. Just add the following `stage` and add the URL you want to test. The source code for this image is available, like our other Jenkins images from the Red Hat CoP.[47] The ZAP scan in Jenkins will produce a report showing some potential vulnerabilities in our application.

```
stage(' OWASP Scan') {
  agent { label "jenkins-agent-zap" }
  steps {
    sh '''
      /zap/zap-baseline.py -r index.html -t http://<some website url> ||
return_code=$?
      echo "exit value was  - " $return_code
    '''
  }
  post {
    always {
      // publish html
      publishHTML target: [
          allowMissing: false,
          alwaysLinkToLastBuild: false,
          keepAll: true,
          reportDir: '/zap/wrk',
          reportFiles: 'index.html',
          reportName: 'OWASP Zed Attack Proxy'
        ]
    }
  }
}
```

46 https://www.zaproxy.org/

47 https://github.com/redhat-cop/containers-quickstarts/tree/master/jenkins-agents

In doing so, the web report that's created can be viewed in Jenkins, which gives great details on the cause of the security vulnerability as well as any action that should be taken to remedy it.



Back to master index Zip

ZAP Scanning Report

Summary of Alerts

Risk Level	Number of Alerts
High	0
Medium	2
Low	5
Informational	4

Alert Detail

Medium (High)	Content Security Policy (CSP) Header Not Set
Description	Content Security Policy (CSP) is an added layer of security that helps to detect and mitigate certain types of attacks, including Cross Site Scripting (XSS) and data injection attacks. These attacks are used for everything from data theft to site defacement or distribution of malware. CSP provides a set of standard HTTP headers that allow website owners to declare approved sources of content that browsers should be allowed to load on that page — covered types are JavaScript, CSS, HTML frames, fonts, images and embeddable objects such as Java applets, ActiveX, audio and video files.
URL	https://pet-battle-labs-test.apps.hivec.sandbox1405.opentlc.com/
Method	GET
URL	https://pet-battle-labs-test.apps.hivec.sandbox1405.opentlc.com/sitemap.xml
Method	GET
Instances	2
Solution	Ensure that your web server, application server, load balancer, etc. is configured to set the Content-Security-Policy header, to achieve optimal browser support: "Content-Security-Policy" for Chrome 25+, Firefox 23+ and Safari 7+, "X-Content-Security-Policy" for Firefox 4.0+ and Internet Explorer 10+, and "X-WebKit-CSP" for Chrome 14+ and Safari 6+.

Figure 15.27: Example Zap report for PetBattle

In the final non-functional testing section, let's have a look at deliberately breaking our code using a technique called chaos engineering.

Chaos Engineering

Chaos engineering is the process of deliberately breaking, hobbling, or impacting a system to see how it performs and whether it recovers in the ensuing "chaos." While most testing is seen as an endeavor to understand how a system performs in a known, stable state, chaos engineering is the computing equivalent of setting a bull free in a fine-china shop—you know it's going to end badly but you just don't know exactly the magnitude of how bad it's going to be.

The purpose of chaos engineering is to build confidence in the resiliency of the system. It also allows you to better understand where breakage points occur and the blast radius of any failures. There are many resilience features built into the Kubernetes API specification. Pod replicas are probably the simplest mechanism, having more than one of your applications running at any given time. It is also desirable to use application-specific mechanisms such as circuit breakers, which prevent failures from spreading throughout your system. Chaos engineering takes these ideas one step further and tests a system when one or more components fully or partially fail, such as when CPU or memory resources are low.

The basic premise is that the system under test is observed in a stable working state, then a fault is injected. The system is then observed to see if it recovers successfully from the fault or not. Outcomes from such testing are a potential list of areas to tune/fix as well as an understanding of the **Mean Time to Recovery** (**MTTR**) of a system. It's important to note that chaos engineering is focused on the system as a whole—both application and infrastructure performance need to be considered and tested.

One of the key mantras behind chaos engineering is contained in its defining principles[48] – *The need to identify weaknesses before they manifest in system-wide, aberrant behaviors.*

This is one of the most important aspects to be considered when adopting this approach. You don't want to be learning about weaknesses during a production-impacting incident. It's similar to the rationale behind regularly testing disaster recovery plans. To paraphrase, a colleague of ours here at Red Hat said, "*When the excrement hits the fan, the first thing to do is turn off the fan!*" Not much time for learning there.

There are a number of tools and frameworks that can help with setting up a chaos engineering practice. Here's some to get started with (though there are others):

- Litmus Chaos[49]
- Kraken[50]
- Chaos Mesh[51]

48 https://principlesofchaos.org/

49 https://litmuschaos.io/

50 https://github.com/cloud-bulldozer/kraken

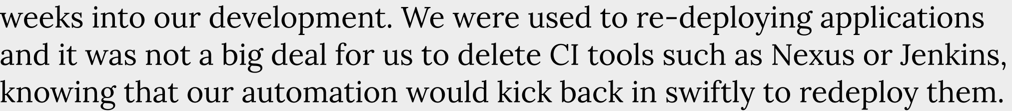
51 https://chaos-mesh.org/

In a world where practices such as everything-as-code and GitOps are our only way to build software and the systems that support them, a great way to validate the ability to respond to missing items is to redeploy everything, including your infrastructure, from scratch every week or every night! This might seem extreme, but it's a great way to validate that there is no hidden magic that someone has forgotten to write down or codify.

Accidental Chaos Testing

This is a story that I used to be reluctant to share, but over the years (and having done it twice), I realized it was actually a good thing to have done.

While working for an airline, I accidentally deleted the `labs-ci-cd` project along with a few other namespaces where our apps were deployed, including the authentication provider for our cluster. At the time, we were several weeks into our development. We were used to re-deploying applications and it was not a big deal for us to delete CI tools such as Nexus or Jenkins, knowing that our automation would kick back in swiftly to redeploy them.



However, on this engagement, we were also using GitLab and, unfortunately for me, GitLab was in the same project as these other tools!

I checked with the team first and asked whether it was OK to rebuild everything in our tooling namespace. I got a resounding "yes" from my teammates, so proceeded to delete some of the things I thought needed to be cleared out and accidentally removed a few extra projects. About 30 seconds later, someone on the team perked up and asked, *Is Git down for anyone else*? This was promptly followed by another person saying, *Is anyone else not able to log in to the cluster*? My face lit up red as I immediately realized what I'd just done. Git, as we keep saying in the book, is our single source of truth. *If it's not in Git, it's not real* is our mantra! We even had it written on the walls! But I had just deleted it.

So, what happens when some silly person accidentally deletes it? After the initial shock and panic, the team pulled the Andon Cord. We quickly stormed together to see what exactly had happened in order to plan how we could recover not just Git but all the things we'd added to the cluster. Luckily for us, everything we had done was stored in Git so we were able to redeploy our tools and push our local, distributed copies of the software and infrastructure back into the shared Git repository.

The team was cross-functional and had all the tools and access we needed to be able to respond to this. Within 1 hour, we had fully restored all our applications and tools with all of our automation running smoothly again.

I think the real power in this example is how, given the right equipment and the right ownership, an empowered team can have it all. We acted as one unit fixing things at lightning speed. We were not stuck waiting in a queue or having to raise a ticket on another team to restore our infrastructure. We could do it for ourselves within minutes – not days or weeks later.

Another thing I learned was not to keep Git in the same project as the other tools in case another person like me comes along. I also learned to be mindful of the permissions we have within a cluster. As an administrator, I was able to remove things that perhaps I should not have been playing with.

So we've written the code, tested, quality-checked it and even scanned it for vulnerabilities. Now it's time to deploy it onto the cluster. Let's explore one of the key areas of benefit of using Kubernetes - the different ways you can deploy applications depending on your needs and perform user-driven experiments to determine what features your users prefer.

Advanced Deployments

The time between software being written and tested till it is deployed in production should be as short as possible. That way your organization is able to realize value from the software changes as quickly as possible. The modern approach to this problem is, of course, through automation. There are simply too many details and configuration items that need to be changed when deploying to production that even for a small application suite like PetBattle, manual deployment becomes error-prone and tedious. This drive to reduce manual toil is at the heart of many of the DevOps practices we have been discovering in this book.

We can minimize the downtime (ideally to zero!) during software deployment changes by adopting the right application architecture and combining that with the many platform capabilities that OpenShift offers. Let's look at some common deployment strategies that OpenShift supports:

- Rolling deployment:
  - Spin up a pod of the new version and then spin down a pod of the existing old version automatically. Very useful for a zero-downtime approach.
- Canary deployment:
  - Spin up a single pod of the new version, perform testing to ensure that everything is working correctly, and then replace all the old pods with new ones.
- Blue/Green deployment:
  - Create a parallel deployment and verify that everything is working correctly before switching traffic over.
  - Service Mesh traffic mirroring functionality can be useful with this approach to validate that the new version is working as expected.
- Recreate deployment:
  - Basically, scale the existing pods down to zero and then spin up the new version.
  - Use where an application must be restarted, for example, to migrate database schema or tables.
  - Think of this as a Ripley deployment: "take off and nuke the entire site from orbit. It's the only way to be sure."[52]

We can roll back to previous deployment versions using the Helm chart life cycle or the out-of-the-box `oc rollback` support. Images and configuration are versioned and cached in OpenShift to easily support rolling back to previous versions.

52 https://en.wikiquote.org/wiki/Aliens_(film)

A/B Testing

A/B testing an application is an amazing way to test or validate a new feature in production. The process is pretty simple: you deploy two (or more) different versions of your application to production, measure some aspect, and see which version performs *better*. Given that A/B testing is primarily a mechanism of gauging user experience, *better* depends on what aspect/feature you're experimenting with. For example, you could make a subtle change to a web page layout and measure how long it takes for the user to navigate to some button or how long the user continues to interact with specific items on the page.

It's a brilliant way to de-risk a new release or validate some new business or UI features with a smaller audience before releasing to a wider group. User behavior can be captured and experiments can be run to make informed decisions about what direction a product should take.

The Experiment

Let's cast our minds back to the earlier chapters where we spoke about generating options. There we spoke about the importance of experiments and our Value Slicing board included an item for which we could do an A/B test. One experiment that came up was to assess how users would vote for cats in the competition. Should they just be able to upvote (with a 👍) or should they be able to downvote (👎) too? We can build and deploy two versions of our application: one with the ability to both upvote and downvote, and one with just the ability to upvote. Our experiment is simple: to track how often people actually use the downvote button, so we can decide whether it's a feature we need or whether we should focus on building different functionality.

Figure 15.28: Experiment defined on a Value Slicing board

Let's now look at how we could set up a simple experiment to deploy both variants of the application and route traffic between each deployed instance to generate some data to help inform our decision-making.

Matomo – Open Source Analytics

OpenShift provides us with a mechanism to push traffic to different versions of an application. This in itself is useful, but it provides no information that we can base a decision on. For this, we need to measure how the users interact with the platform. To do this, we're going to introduce user analytics, which records metrics on the users' interactions with the website. We're going to use the open source Matomo[53] platform. There are others we could have used but, at the time of writing, this was our choice as it was open source and quite feature-complete. Let's add Matomo to our Big Picture for consistency.

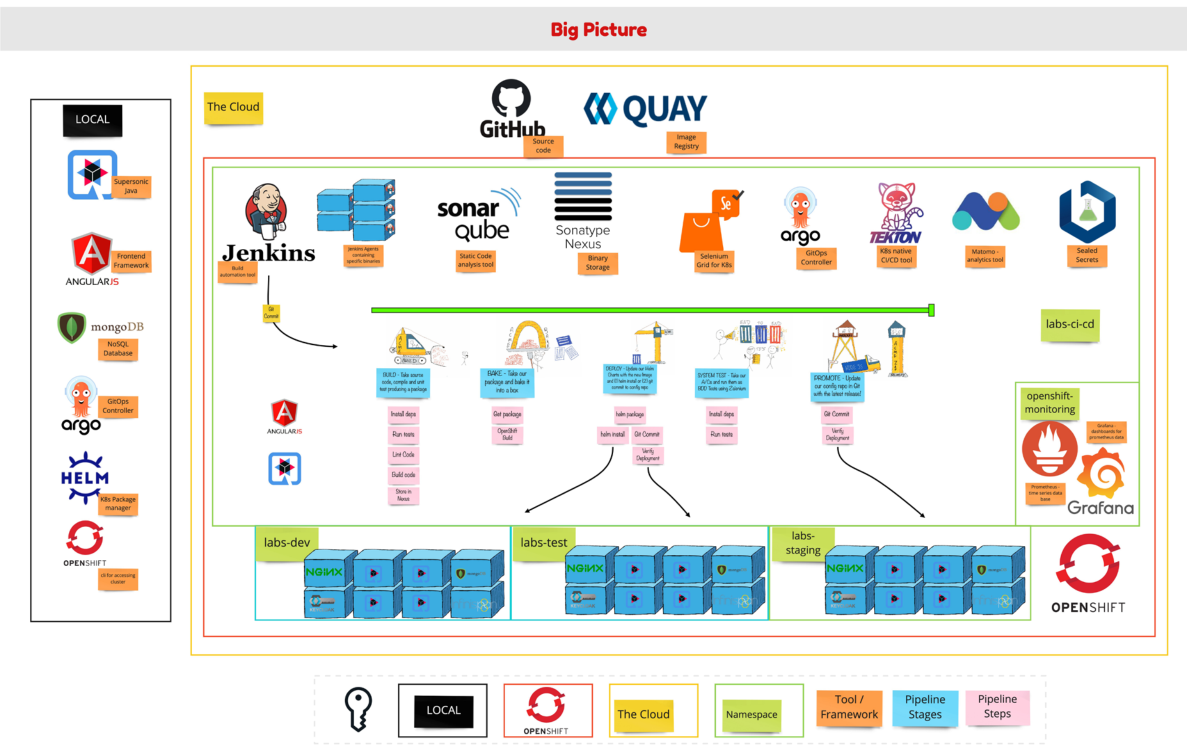


Figure 15.29: Big Picture with added tools including Matomo

53 https://matomo.org/

So how do we install the Matomo platform? Here comes Helm to the rescue again. We automated this installation as part of the PetBattle platform by just enabling it in our Ubiquitous Journey project. It's deployed by default into our `labs-ci-cd` namespace from this configuration in `ubiquitous-journey/values-tooling.yaml`:

```
# Matamo
  - name: matomo
    enabled: true
    source: https://github.com/petbattle/pet-battle-analytics.git
    source_path: charts/matomo
    sync_policy: *sync_policy_true
    destination: labs-ci-cd
    source_ref: main
    ignore_differences:
    - group: apps
      kind: Deployment
      jsonPointers:
        - /spec/replicas
        - /spec/template/spec/containers/0/image
```

However, if you want to just install the tool without involving ArgoCD, you can just clone the repository and install it manually. This chart has been forked from an existing chart[54] to tweak it for easier installation on OpenShift. Specifically, the security contexts in the MariaDB and Redis dependencies have been disabled so that the deployment will automatically use the target namespace default service account and associated **Security Context Constraint** (**SCC**) in OpenShift. For COTS software where repackaging or running as a random UID is not always possible, there are other more permissive, less secure SCCs such as `anyuid`. Also, an OpenShift route has been added to the chart to allow ingress traffic to the application:

```
$ oc login ...
$ git clone https://github.com/petbattle/pet-battle-analytics.git \
&& cd pet-battle-analytics
$ helm install pba charts/matomo
```

With the Matomo analytics deployed, we just need to configure the frontend to connect to it. To do this just update the config map's `matomoUrl` in the `chart/values.yaml` in the frontend to have the tracking code automatically track the site. This will provide basic site tracking such as the time spent on a page or the number of pages visited.

54 https://gitlab.com/ideaplexus/helm/matomo

```
23  # custom end point injected by config map. This is likely to changed
24  config_map: '{
25    "catsUrl": "https://pet-battle-api-labs-test.apps.hivec.sandbox1405.opentlc.com",
26    "tournamentsUrl": "https://pet-battle-tournament-labs-test.apps.hivec.sandbox1405.opentlc.com",
27    "matomoUrl": "https://matomo-labs-ci-cd.apps.hivec.sandbox1405.opentlc.com/",
28    "keycloak": {
29      "url": "https://keycloak-labs-test.apps.hivec.sandbox1405.opentlc.com/auth/",
30      "realm": "pbrealm",
31      "clientId": "pbclient",
32      "redirectUri": "http://localhost:4200/tournament",
33      "enableLogging": true
34    }       You, 3 months ago • Config load correction and sso stuff oh and co…
35  }'
36
```

Figure 15.30: Configuring the config_map for matomoUrl

For a more meaningful test, we might want to capture specific user behavior. The application has been instrumented to report certain events back to the Matomo server, such as mouse clicks. Whenever a user clicks the button to vote for a cat, it will capture it and report it in Matomo for us. It's very simple to do this – we just add a one-liner to the event we want to track:

```
this.matomoTracker.trackEvent('A/B Tests', 'Voting', voting)
```

Deploying the A/B Test

In PetBattle land, let's see how we could configure the deployments of the frontend to run this simple A/B test. Luckily for us, OpenShift makes this super easy by having a way to expose a `route` and connect it to more than one service, using the `alternateBackends` array to configure additional services to send traffic to. We can then apply `weights` to each service defined here in order to set the percentage of the traffic to either service that's deployed, A or B. The weights can be set between 0 and 256, and if a service is reduced to 0 then it carries on serving existing connections but no new ones. In fact, OpenShift allows us to do more than just an A or B test – also C and D, as `alternateBackends` supports up to three services!

Let's deploy our A/B experiment for `pet-battle`. We could integrate these steps with ArgoCD but to keep things nice and easy for illustrative purposes, let's just stick with using our trusty friend Helm to deploy things. We've prebuilt an image that has no ability to downvote on the home page `quay.io/petbattle/pet-battle:no-down-vote`. Let's deploy this image to our cluster by running a simple Helm command (make sure to set the config map to the correct endpoints for your cluster):

```
$ git clone git@github.com:petbattle/pet-battle.git && cd pet-battle

$ helm install nodownvote --set image_version=no-down-vote \
--set route=false chart --namespace petbattle
```

With this command, we're deploying a new instance of the `pet-battle` frontend by setting the image to the prebuilt one and disabling the OpenShift route for this as it's not needed. We'll configure our route to production by updating our `prod` app.

Running `oc get pods` should show the app started and if you check for routes, you should see none exposed:

```
$ oc get pods

NAME                             READY   STATUS      RESTARTS   AGE
nodownvote-pet-battle-1-deploy   0/1     Completed   0          2m47s
nodownvote-pet-battle-1-jxzhf    1/1     Running     0          2m43s
```

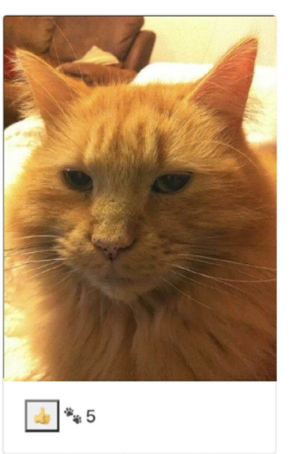
Let's deploy our `prod` version of the `pet-battle` application and add the `no-down-vote` app as one of the services we'll connect to. Our Helm chart is configured to accept the name of the service and the weight we want to apply to the experiment feature via `a_b_deploy.svc_name` and `a_b_deploy.weight`. It's defaulted to be a 50/50 round-robin split. Let's deploy it with this setup:

```
# install prod version
$ helm install prod --set image_version=latest chart \
--set a_b_deploy.svc_name=no-down-vote-pet-battle --namespace petbattle

# list pods
$ oc get pods

NAME                             READY   STATUS              RESTARTS   AGE
nodownvote-pet-battle-1-deploy   0/1     Completed           0          4m53s
nodownvote-pet-battle-1-jxzhf    1/1     Running             0          4m49s
prod-pet-battle-1-6bbv8          0/1     ContainerCreating   0          12s
prod-pet-battle-1-deploy         1/1     Running             0          16s
```

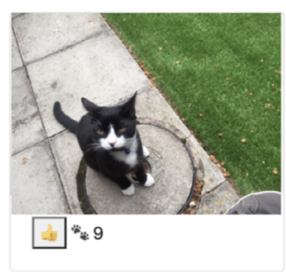
Navigate to the `pet-battle` UI and you should see on refreshing that there is a 50/50 chance that you will get the upvote-only version. If you open up incognito mode or a different browser and try to hit the frontend, you should get the alternative one. A different browser session is required, as the OpenShift router will by default return you to the same pod, so you'll always land on the same site version.

Refresh the Leaderboard

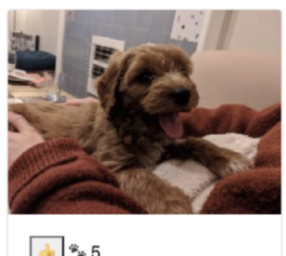
Upvote or Downvote your favourite cat to reveal another one ...



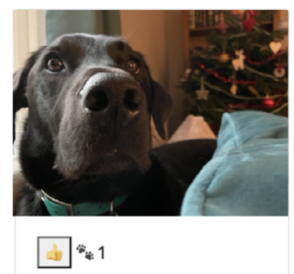
5



9



5



1

Figure 15.31: The no-downvote PetBattle frontend

Running `oc get routes` should show one route and more than one service connected to it with a 50/50 split `prod-pet-battle(50%),no-down-vote-pet-battle(50%)`. You can view the weights set as 100 each by running `oc get route prod-pet-battle -o yaml`:

```
# display the routes
$ oc get routes

NAME        HOST/PORT      PATH   SERVICES  PORT  TERMINATION  WILDCARD
prod-pet-battle       prod-pet-battle-labs-dev.apps…
      prod-pet-battle(50%),no-down-vote-pet-battle(50%)
             8080-tcp   edge/Redirect   None
```

The weights for the traffic routed to each application can be updated quite easily using Helm:

```
# update route weights
$ helm upgrade prod --set image_version=latest chart \
    --set a_b_deploy.svc_name=no-down-vote-pet-battle \
    --set a_b_deploy.weight=10 --namespace petbattle
```

Understanding the results

If we play around with the two versions that are deployed, we can see how the results of clicking the buttons are captured. If you open the Matomo app and log in, you will see some statistics there. The default password for Matomo, as set in the chart, is `My$uper$ecretPassword123#`. This might not be exactly secure out of the box but it can easily be changed via the Helm chart's values.

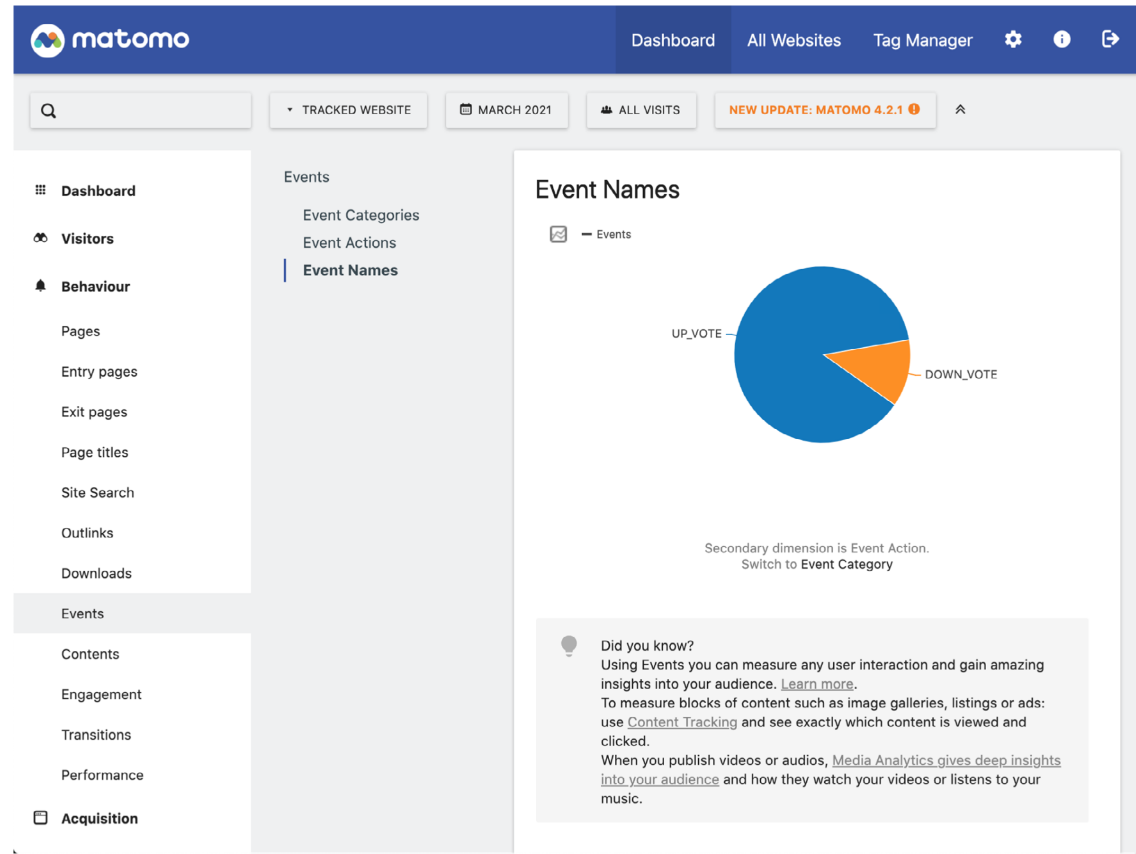


Figure 15.32: Matomo showing the number of clicks for UP_VOTE versus DOWN_VOTE

It might take a few minutes for Matomo to render the pie chart. Our simple experiment shows that more people use the **UP_VOTE** feature than the **DOWN_VOTE** feature. By connecting the A/B test to the data captured in Matomo, we can now make more informed decisions about the next actions that need to be taken for our product.

This experiment proves how easy it is to set up an A/B test. We can use the OpenShift platform to dynamically route users to multiple application versions concurrently deployed while we collect data about what is working well and what is not. There is some thinking that needs to be put into how we instrument the application to collect specific data, but the open source tooling available to us makes this easy too!

Blue/Green deployments

The Blue/Green deployment strategy is one of the fundamental deployment strategies that every team deploying applications into production should know about. Using this strategy minimizes the time it takes to perform a deployment cutover by ensuring you have two versions of the application available during deployment. It is also advantageous in that you can quickly roll back to the original version of the application without having to roll back any changes.

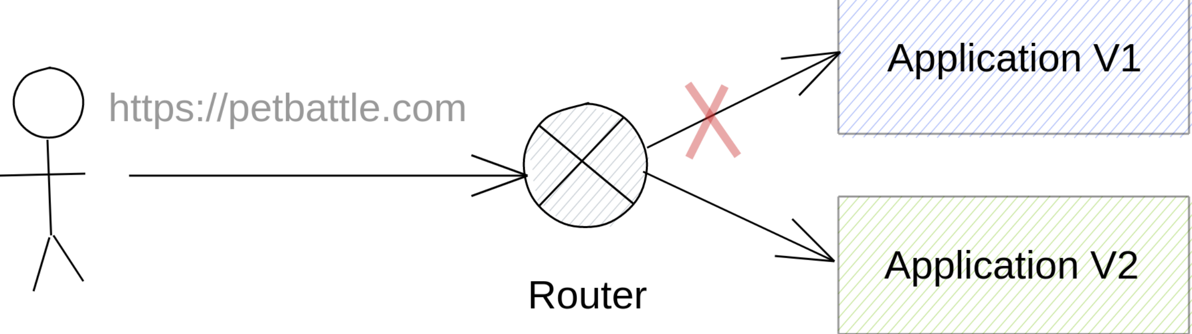


Figure 15.33: The canonical Blue/Green deployment

The trade-off here is that you need to have enough resources to be able to run two versions of the application stack you are deploying. If your application has persistent state, for example, a database or non-shared disk, then the application architecture and constraints must be able to accommodate the two concurrent versions. This is normally not an issue for smaller microservices and is one of the benefits of choosing that style of deployment.

Let's run through Blue/Green deployment using the PetBattle API as the example application stack. In this case, we are going to deploy two full stacks, that is, both the application and MongoDB. Let's deploy the blue version of our application:

```
# install the blue app stack
$ helm upgrade --install pet-battle-api-blue \
    petbattle/pet-battle-api --version=1.0.15 \
    --namespace petbattle --create-namespace
```

Now deploy the green application stack. Note that we have a different tagged image version for this:

```
# install the green app stack
$ helm upgrade --install pet-battle-api-green \
  petbattle/pet-battle-api --version=1.0.15 \
    --set image_version=green \
    --namespace petbattle
```

Next, we expose our production URL endpoint as a route that points to the blue service:

```
# create the production route
$ oc expose service pet-battle-api-blue --name=bluegreen \
--namespace petbattle
```

Finally, we can switch between the two using the `oc patch` command:

```
# switch service to green
$ oc patch route/bluegreen --namespace petbattle -p \
    '{"spec":{"to":{"name":"pet-battle-api-green"}}}'

# switch back to blue again
$ oc patch route/bluegreen --namespace petbattle -p \
    '{"spec":{"to":{"name":"pet-battle-api-blue"}}}'
```

If you browse to the `bluegreen` route endpoint, you should be able to easily determine the application stack:

Figure 15.34: Blue/Green deployment for the PetBattle API

Even though this is somewhat of a contrived example, you can see the power of developers being allowed to manipulate the OpenShift routing tier in a self-service manner. A similar approach could be used to deploy the NSFF feature as an example – use the Helm chart parameters `--set nsff.enabled=true` to deploy an NSFF-enabled version. You can also point both applications to the same database if you want to with similar manipulation of the Helm chart values.

If you have more complex use cases where you need to worry about long-running transactions in the original blue stack, that is, you need to drain them, or you have data stores that need migrating alongside the green rollout, there are several other more advanced ways of performing Blue/Green deployments. Check out the ArgoCD rollout capability, which has a ton of advanced features,[55] the Knative Blue/Green rollout capability, or indeed Istio[56] for more ideas.

Deployment previews

We should think of OpenShift as something of a playground that we can use to deploy our applications for production all the way down to a developer preview. Gone are the days when a development team needed to raise a ticket to provision a server and manually configure it to show off their applications. Building applications in containers allows us to make shippable applications that can be repeatedly deployed in many environments. Our automation for PetBattle in Jenkins is configured to run on every commit. For Jenkins, we're using the multi-branch plugin so anytime a developer pushes a new feature to a branch, it will automatically scaffold out a new pipeline and deploy the latest changes for that feature.

When this was discussed in the previous chapter, about sandbox builds, you may have thought this was overkill and a bit of a waste. Why not just build on a pull request? It's a valid question to ask and depending on the objective you're trying to achieve, building on a pull request is probably sufficient. We have used the sandbox builds as another way to introduce feedback loops.

55 https://argoproj.github.io/argo-rollouts

56 https://github.com/hub-kubernetes/istio-blue-green-deployment

Developers do not exist in isolation; they are surrounded by other members of the team, including Product Owners and Designers. Our ability to dynamically spin up a new deployment of a feature from our pipeline means we can connect the coding efforts to the design team really easily. Developers can get very fast feedback by sharing a link to the latest changes or the implementation of a new feature with the design team. This feedback loop can quickly allow subtle changes and revisions to be made before the engineer loses the context of the piece of work. Creating deployment previews from every commit also allows a developer to very quickly share two versions of what an app might look like with a Product Owner while they make their decision about which to choose.

From our Jenkins pipeline, there is a branch called `cool-new-cat`. When this is built, it will push a new version of the app to the `dev` environment. The change in the app is subtle for illustrative purposes, but we can see the banner has been changed. With this new version of the app in the `dev` environment, we can get some feedback prior to merging it to master and generating a release candidate.

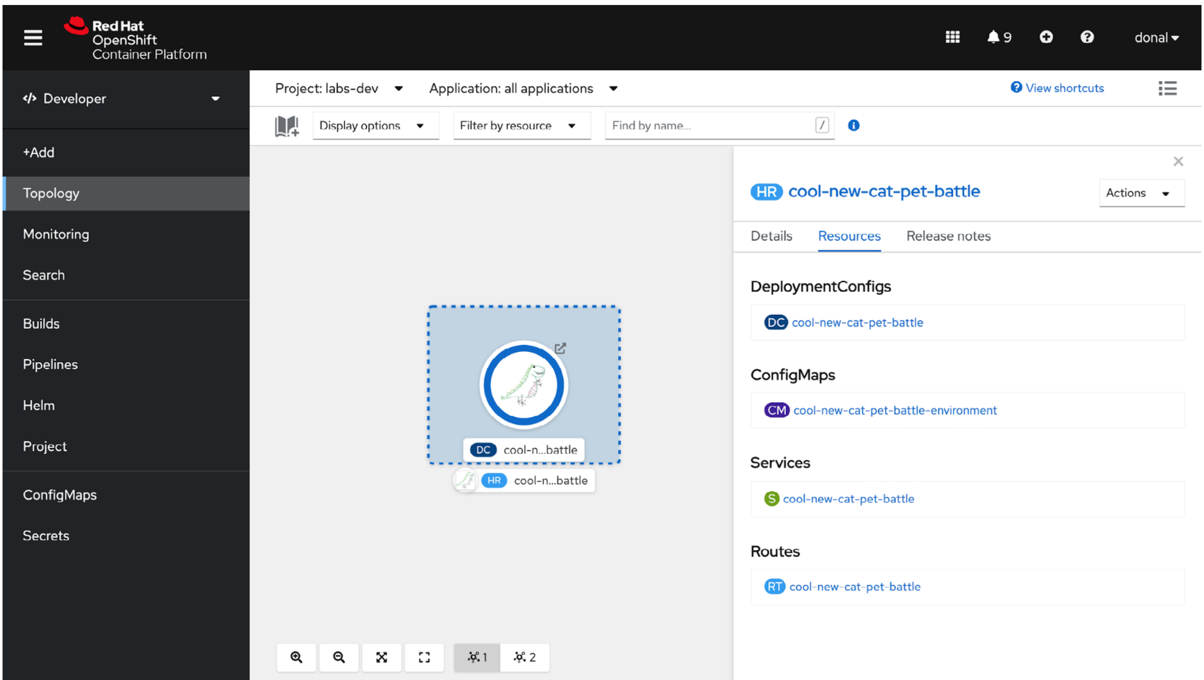


Figure 15.35: New feature deployed to the sandbox generating a deploy preview to collect feedback

Figure 15.35 shows the sandbox version of the being deployed along with it's associated route, service and configmap.

Conclusion

Congratulations! You've just finished the most technologically focused chapter of this book so far. Please don't go off and think that you have to use each and every technology and technique that has been mentioned—that's not the point. Investigate, evaluate, and choose which of these technologies applies to your own use cases and environment.

Several of the testing practices are part of our technical foundation. Unit testing, non-functional testing, and measuring code coverage are all critical practices for helping build quality into our applications and products from the start. We covered many small but invaluable techniques, such as resource validation, code linting, and formatting, that help make our code base less of a burden to maintain.

We covered a number of different approaches for deployments, including A/B, Canary, Blue/Green, and Serverless. These core techniques allow us to deliver applications more reliably into different environments. We even briefly covered artificial intelligence for reducing unwanted images uploaded into our PetBattle product. By focusing our efforts on what happens when things go wrong, we can more easily embrace and prepare for failures—big and small.

16

Own It

"*Annnddd we're live! PetBattle is finally in production, we can crack open the Champagne and toast our success*." But now what? How do we know that the site is doing what we expect it to do and, more importantly, how will we know when it isn't performing as we intended? Do we just sit around waiting for customers to complain that the site is down or that errors are happening? Not exactly a good user experience model—or a good use of our time.

In this chapter, we will discuss the tools and techniques that can be utilized to monitor the site and notify us when things start to go wrong so we can react before the entire site goes down. We will also discuss advanced techniques, such as Operators, that can help you automate a lot of the day-to-day operations.

Observability

Observability[1] is the process of instrumenting software components to assist with extracting data. This data can then be used to determine how well a system is functioning and subsequently be used to notify administrators in the event of issues.

1 https://en.wikipedia.org/wiki/Observability

When it comes to observing the state of our PetBattle applications, there are a number of aspects to consider:

- How do we know if an application instance is initialized and ready to process traffic?
- How do we know if the application has failed without crashing, such as by becoming deadlocked or blocked?
- How do we access the application logs?
- How do we access the application metrics?
- How do we know what version of the application is running?

Let's start by exploring some application health checks.

Probes

Hello, hello?... Is this thing on? In Kubernetes, the health of an application is determined by a set of software *probes* that are periodically invoked by the kubelet. A probe is basically an action invoked by the platform on each Pod that either returns a success value or a failure value.

Probes can be configured to perform one of the following types of actions:

- Connect to a specific TCP port that the container is listening on. If the port is open, the probe is considered successful.
- Invoke an HTTP endpoint, if the HTTP response code is 200 or greater but less than 400.
- Shell into a container and execute a command—this may involve checking for a specific file in the directory. This enables probes to be placed on applications that don't natively provide health checks out of the box. If the command exits with a status code of 0, then the probe is successful.

If a probe fails a configured number of times, the kubelet managing the Pod will take a pre-determined action, for example, by removing the Pod from the service or restarting the Pod.

Kubernetes currently supports three different kinds of probes:

1. **Readiness**: This decides whether the Pod is ready to process incoming requests. If the application needs some time to start up, this probe ensures that no traffic is sent to the Pod until this probe passes. Also, if the probe fails while it's running, the platform stops sending any traffic to the Pod until the probe once again succeeds. Readiness probes are key to ensuring a zero-downtime experience for the user when scaling up or upgrading Pods.

2. **Liveness**: This checks to see whether a Pod has a process deadlock or it's crashed without exiting; if so, the platform will kill the Pod.
3. **Startup**: This is used to prevent the platform from killing a Pod that is initializing but is slow in starting up. When the startup probe is configured, the readiness and liveness probes are disabled until the startup probe passes. If the startup probe never passes, the Pod is eventually killed and restarted.

Most of the time, you will probably only utilize the readiness and liveness probes, unless you have a container that's very slow in starting up.

In the PetBattle Tournament Service component, the liveness and readiness probes are configured as follows.

In `DeploymentConfig` (or the Deployment), the `/health/live` and `/health/ready` URLs are automatically created by the Quarkus framework:

```
...
livenessProbe:
  failureThreshold: 3
  httpGet:
    path: /health/live
    port: 8080
    scheme: HTTP
  initialDelaySeconds: 0
  periodSeconds: 30
  successThreshold: 1
  timeoutSeconds: 10
readinessProbe:
  failureThreshold: 3
  httpGet:
    path: /health/ready
    port: 8080
    scheme: HTTP
  initialDelaySeconds: 0
  periodSeconds: 30
  successThreshold: 1
  timeoutSeconds: 10
```

Different probes can invoke the same action, but we consider this bad practice.
The semantics of a readiness probe are different from those of a liveness probe. It's recommended that liveness and readiness probes invoke different endpoints or actions on the container.

For example, a readiness probe can invoke an action that verifies whether an application can accept requests. If during the Pod's lifetime the readiness probe fails, Kubernetes will stop sending requests to the Pod until the probe is successful again.

A liveness probe is one that verifies whether an application can process a request successfully; for example, if an application were blocked or accepting a request but waiting a long time for a database connection to become available, the probe would fail and Kubernetes would restart the Pod. Think of liveness probes as the Kubernetes equivalent of the IT Crowd[2] way of working.

Domino Effect

One question that we get asked a lot is, should a health check reflect the state of the application's downstream dependencies as well as the application itself? The absolute, definitive answer is *it depends*. Most of the time, a health check should only focus on the application, but there are always scenarios where this isn't the case.

If your health check functionality does a deep check of downstream systems, this can be expensive and result in cascading failures, where a downstream system has an issue and an upstream Pod is restarted due to this downstream issue. Some legacy downstream systems may not have health checks, and a more appropriate approach in this scenario is to add resilience and fault tolerance to your application and architecture.

Fault Tolerance

A key aspect of this is to utilize a **circuit breaker** pattern when invoking dependencies. Circuit breakers can *short circuit* the invocation of downstream systems when they detect that previous calls have failed. This can give the downstream system time to recover or restart without having to process incoming traffic.

The basic premise of a circuit breaker is that in the case of the failure of a downstream system, the upstream system should just assume that the next request will fail and not send it. It potentially also takes appropriate actions for recovery by, say, returning a default value.

After a given period of time, known as the backoff period, the upstream system should try sending a request to the downstream system, and if that succeeds, it reverts to normal processing. The rationale behind the backoff period is to avoid the situation where the upstream systems overwhelm the downstream systems with requests as soon as it starts up.

---

2 https://www.quotes.net/mquote/901983

Circuit breaker functionality can be performed at an individual level within an application's code: multiple frameworks such as Quarkus, Netflix Hystrix, and Apache Camel support circuit breakers and other fault-tolerant components. Check out the Quarkus fault tolerance plugin for more details.[3]

Platform-level circuit breaker functionality is provided by the **service mesh** component within OpenShift. This has various substantial advantages over application-level circuit breakers:

- It can be used on any container communicating via HTTP/HTTPS. A sidecar proxy is used to inject the circuit breaker functionality without having to modify the code.
- It provides dynamic configuration of the circuit breaker functionality.
- It provides metrics and visibility of the state of circuit breakers throughout the platform.
- Service mesh also provides other fault-tolerance functionalities, such as timeout and retries.

Logging

Ahh, logging! No *true* developer[4] has earned their stripes until they've spent countless hours of their existence trawling through production logs trying to figure out exactly what went wrong when a user clicked "Confirm". If you have managed to do this across multiple log files, all hosted on separate systems via multiple terminal windows, then you are truly righteous in the eyes of the IDE-bound masses.

The good news is that application logging on Kubernetes is a first-class citizen on the platform—just configure your application to write its logs to STDOUT and the platform will pick it up and you can view/trawl through them. OpenShift goes one level deeper by shipping an aggregated logging stack with EFK (Elasticsearch, Fluentd, and Kibana) out of the box. This allows developers to search and view logs across multiple containers running on multiple nodes across the cluster. If you want to give this a try, follow the documentation at https://docs.openshift.com/container-platform/4.7/logging/cluster-logging-deploying.html.

3 https://quarkus.io/guides/smallrye-fault-tolerance

4 https://en.wikipedia.org/wiki/No_true_Scotsman

Tracing

So, first things first: no, tracing is not application logging running at the trace log level. When it comes to OpenShift, tracing is the functionality added to the Kubernetes platform that enables developers to trace a request across a distributed set of application components running in different containers on different nodes of a cluster. Tracing is an exceptionally useful tool used to determine and visualize inter-service/component dependencies and performance/latency blackholes throughout a distributed system.

Tracing is provided as part of the OpenShift service mesh component. The underlying tracing functionality is provided by the Jaeger[5] distributed tracing platform. To support tracing, applications must include a client library that sends instrumented request metadata to a Jaeger collector, which in turn processes it and stores the data. This data can then be queried to help visualize the end-to-end request workflow. The Jaeger client libraries are language-specific and utilize the vendor-neutral OpenTracing specification.

If you're thinking, "*Woah! Collecting metadata for every request would be very expensive to store and process*," you'd be right. Jaeger *can* do this, but for scale purposes it's better to record and process a *sample* of requests, rather than each and every one of them.

Metrics

Probes are useful for telling when an application is ready to accept traffic, or whether it is stuck. Tracing is great at providing a measure of latency throughout a distributed system, while logging is a great tool to retrospectively understand exactly what occurred and when it occurred.

However, to comprehend the deep state (no, not *that* deep state!) of a system and potentially predict its future state after a period of time, you need to measure some of the key quantitative characteristics of the system and visualize/compare them over a period of time.

The good news is that metrics are relatively easy to obtain; you can get them from infrastructure components and software components such as JVM, and you can also add domain-specific/custom metrics to your application.

5 https://www.jaegertracing.io/

Given the multitude of metrics available, the hard bit is figuring out which metrics are valuable to your role and need to be retained; for example, for application operator connection pool counts, JVM garbage collection pause times are invaluable. For a Kubernetes platform operator, JVM garbage collection pause times are less critical, but metrics from platform components, such as etcd-related metrics, are crucial.

The good news is that OpenShift provides metrics for both the cluster and the applications running on it. In this section, we're going to focus on the application-level perspective. In the Kubernetes community, the *de facto* approach is to use Prometheus[6] for gathering and Grafana[7] for the visualization of metrics. This doesn't mean that you can't use other metrics solutions, and there are some very good ones out there with additional features.

OpenShift ships with both Prometheus and Grafana as the default metrics stack. Additionally, it also ships with the Prometheus Alertmanager. The Alertmanager facilitates the sending of notifications to operators when metric values indicate that something is going or has gone wrong and *la merde* has or is about to hit the fan. Examples of this include a high number of threads or large JVM garbage collection pause times.

Great, so how do we enable this for PetBattle? It is relatively straightforward:

1. Use a metrics framework in your application that records metrics and exposes the metrics to Prometheus.
2. Configure Prometheus to retrieve the metrics from the application.
3. Visualize the metrics in OpenShift.

Once the metrics are being retrieved, the final step is to configure an alert using the Prometheus Alertmanager.

Gather Metrics in the Application

Taking the PetBattle Tournament service component as an example, this is developed using the Quarkus Java framework. Out of the box, Quarkus supports/recommends the use of the open-source Micrometer metrics framework.[8]

---

6 https://prometheus.io/

7 https://grafana.com/oss/

8 https://micrometer.io/

To add this to the Tournament service, we simply need to add the dependency to the Maven POM along with the Prometheus dependency. For example:

```
<dependency>
   <groupId>io.quarkus</groupId>
   <artifactId>quarkus-micrometer</artifactId>
</dependency>
<dependency>
   <groupId>io.micrometer</groupId>
   <artifactId>micrometer-registry-prometheus</artifactId>
</dependency>
```

Then, we configure a Prometheus registry, which is used to store the metrics locally in the application before being retrieved by the Prometheus collector. This is done in the `src/main/resources/application.properties` file.

```
# Metrics
quarkus.micrometer.enabled=true
quarkus.micrometer.registry-enabled-default=true
quarkus.micrometer.binder-enabled-default=true
quarkus.micrometer.binder.jvm=true
quarkus.micrometer.binder.system=true
quarkus.micrometer.export.prometheus.path=/metrics
```

With this configuration, the Prometheus endpoint is exposed by the application Pod. Let's go ahead and test it:

```
# grab the pod name for the running tournament service
$ oc get pod -n petbattle | grep tournament
$ oc exec YOUR_TOURNAMENT_PODNAME -- curl localhost:8080/metrics
...
# HELP mongodb_driver_pool_size the current size of the connection pool,
including idle and and in-use members
# TYPE mongodb_driver_pool_size gauge
mongodb_driver_pool_size{cluster_id="5fce8815a685d63c216022d5",server_
address="my-mongodb:27017",} 0.0
# TYPE http_server_requests_seconds summary
http_server_requests_seconds_
count{method="GET",outcome="SUCCESS",status="200",uri="/openapi",} 1.0
http_server_requests_seconds_
sum{method="GET",outcome="SUCCESS",status="200",uri="/openapi",} 0.176731581
http_server_requests_seconds_count{method="GET",outcome="CLIENT_
ERROR",status="404",uri="NOT_FOUND",} 3.0
http_server_requests_seconds_sum{method="GET",outcome="CLIENT_
ERROR",status="404",uri="NOT_FOUND",} 0.089066563
```

```
http_server_requests_seconds_
count{method="GET",outcome="SUCCESS",status="200",uri="/metrics",} 100.0
# HELP http_server_requests_seconds_max
# TYPE http_server_requests_seconds_max gauge
http_server_requests_seconds_
max{method="GET",outcome="SUCCESS",status="200",uri="/openapi",} 0.176731581
http_server_requests_seconds_max{method="GET",outcome="CLIENT_
ERROR",status="404",uri="NOT_FOUND",} 0.0
...
```

If successful, you should get an output similar to the above. Notice that you're not just getting the application-level metrics—the MongoDB connection pool metrics are also there. These are automatically added by the Quarkus framework once configured in the `application.properties` file.

Configuring Prometheus To Retrieve Metrics From the Application

Prometheus is somewhat unusual in its mode of operation. Rather than having some sort of agent pushing metrics to a central collector, it uses a pull model where the collector retrieves/scrapes metrics from a known HTTP/HTTPS endpoint exposed by the applications. In our case, as seen above, we're exposing metrics using the `/metrics` endpoint.

So how does the Prometheus collector know when, where, and how to gather these metrics? OpenShift uses a Prometheus operator[9] that simplifies configuring Prometheus to gather metrics. We just need to deploy a `ServiceMonitor` object to instruct Prometheus on how to gather our application metrics.

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
 labels:
   app.kubernetes.io/component: pet-battle-tournament
   k8s-app: pet-battle-tournament
 name: pet-battle-tournament-monitor
spec:
 endpoints:
 - interval: 30s
   port: tcp-8080
   scheme: http
 selector:
   matchLabels:
     app.kubernetes.io/component: pet-battle-tournament
```

9 https://github.com/prometheus-operator/prometheus-operator

There are a few things to note that might save you some time when trying to understand this configuration: basically, this configuration will scrape associated components every 30 seconds using the default HTTP path `/metrics`. Now, `port: tcp-8080` is mapped to the port name in the service—see below, highlighted in bold. If the service had a port name of `web`, then the configuration would be `port: web`.

```
$ oc describe svc my-pet-battle-tournament
Name:                my-pet-battle-tournament
Namespace:           pet-battle-tournament
Labels:              app.kubernetes.io/component=pet-battle-tournament
                     app.kubernetes.io/instance=my
                     app.kubernetes.io/managed-by=Helm
                     app.kubernetes.io/name=pet-battle-tournament
                     app.kubernetes.io/version=1.0.0
                     deploymentconfig=my-pet-battle-tournament
                     helm.sh/chart=pet-battle-tournament-1.0.0
Annotations:         Selector:  app.kubernetes.io/component=pet-battle-
tournament,app.kubernetes.io/instance=my,app.kubernetes.io/name=pet-battle-
tournament,deploymentconfig=my-pet-battle-tournament
Type:                ClusterIP
IP:                  172.30.228.67
Port:                tcp-8080  8080/TCP
TargetPort:          8080/TCP
Endpoints:           10.131.0.28:8080
Port:                tcp-8443  8443/TCP
TargetPort:          8443/TCP
Endpoints:           10.131.0.28:8443
Session Affinity:  None
Events:              <none>
```

User workload monitoring needs to be enabled at the cluster level before ServiceMonitoring will work.[10] This is also a classic demonstration of two of the major, powerful, misunderstood, and unused features of Kubernetes, *labels* and *label selectors*.

The following line means that Prometheus will attempt to retrieve metrics from all components that have the label `app.kubernetes.io/component: pet-battle-tournament`. We don't need to list each component independently; we just need to ensure that the component has the correct *label*, and that the *selector* is used to match that label. If we add a new component to the architecture, then all we have to do is ensure that it has the correct label. Of course, all of this assumes that the method of

10 https://docs.openshift.com/container-platform/4.7/monitoring/enabling-monitoring-for-user-defined-projects.html

scraping the metrics is consistent across all of the selected components; that they are all using the `tcp-8080 port`, for example.

```
selector:
  matchLabels:
    app.kubernetes.io/component: pet-battle-tournament
```

We're big fans of labels and their associated selectors. They're very powerful as a method of grouping components: Pods, Services, and so on. It's one of those hidden gems that you wish you knew of earlier.

Visualizing the Metrics in OpenShift

Once we have our metrics retrieved, we need to interpret the information that they're conveying about the system.

Querying using Prometheus

To visualize the metrics, go into the developer console and click on **Monitoring (1)**, as shown in *Figure* 16.1. Then click on **Custom Query (2)** in the dropdown, and enter a query using the **Prometheus query language (PromQL) (3)**. In the following example, we've used the *http_server_requests_seconds_count* metric, but there are others as well.

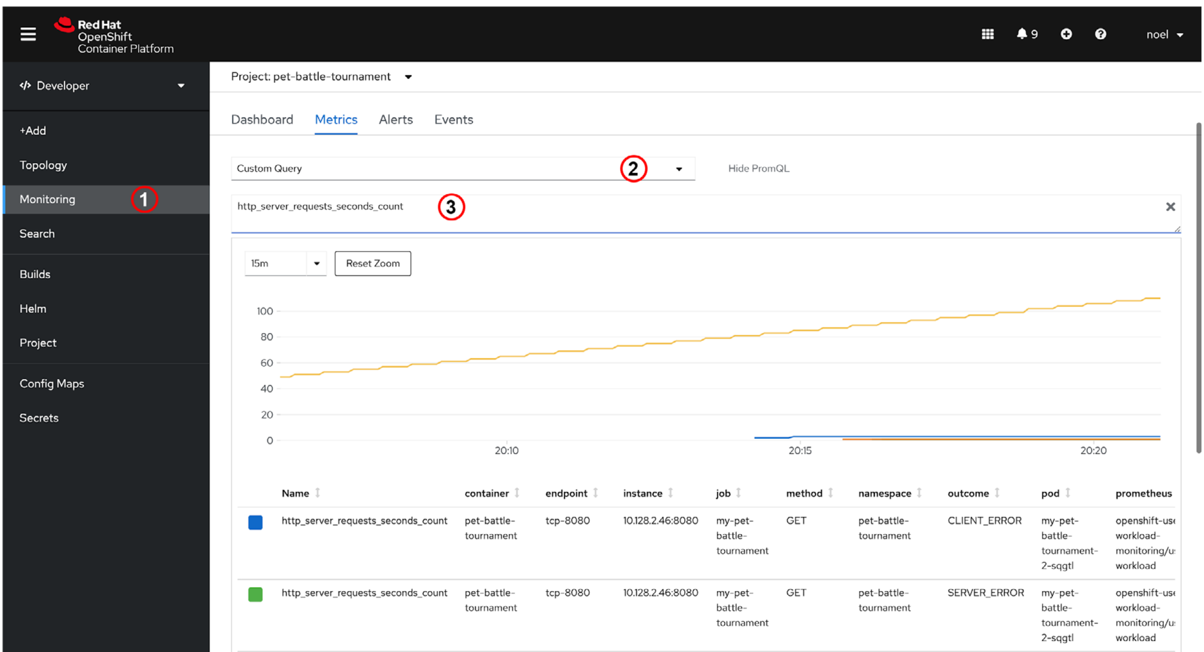


Figure 16.1: PetBattle tournament metrics

Let's explore some of the built-in dashboards OpenShift provides for monitoring.

Visualizing Metrics Using Grafana

OpenShift comes with dedicated Grafana dashboards for cluster monitoring. It is not possible to modify these dashboards and add custom application metrics, but it is possible to deploy an application-specific Grafana instance and customize that as we see fit. To do this, we first need to ensure that the Grafana Operator is installed in the namespace that we're using.

We will then deploy a custom Grafana setup by deploying the following custom resources:

- A *Grafana resource* used to create a custom *grafana* instance in the namespace
- A *GrafanaDataSource resource* to pull metrics from the cluster-wide Prometheus instance
- A *GrafanaDashboard* resource for creating the dashboard

The good news is that all of this is done via Helm charts, so you just have to do the following:

```
$ oc get routes
...
grafana-route     grafana-route-pb-noc.apps.someinstance.com
```

Open `grafana-route` in a browser, log in, *et voila*! It should look something like that shown in *Figure 16.2*. If there is an error with no data, check the BEARER_TOKEN is in place. This can be fixed manually by running the commands at https://github.com/petbattle/pet-battle-infra/blob/main/templates/insert-bearer-token-hook.yaml#L80

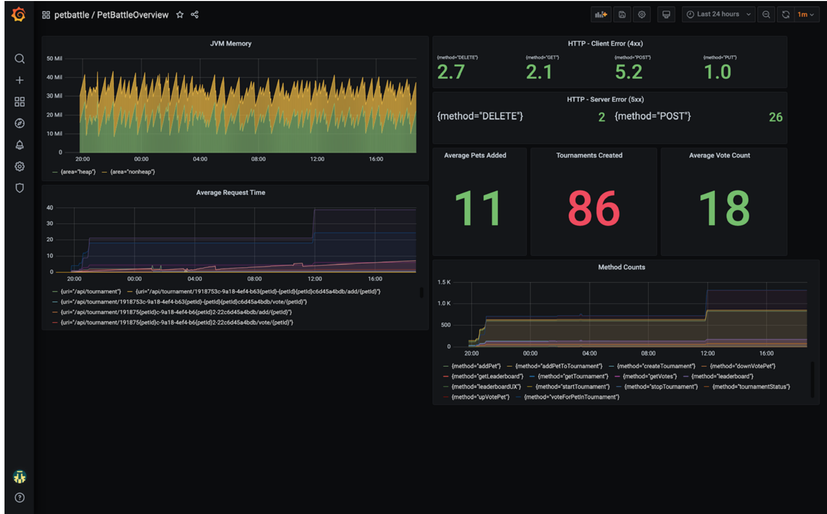


Figure 16.2: PetBattle metrics in Grafana

We will now take a look at some of the tools that can help us further with observability.

Metadata and Traceability

With the adoption of independently deployable services-based architectures, the complexity of managing these components and their inter-relationships is becoming problematic. In the following sections, we will outline a number of techniques that can assist you with this.

Labels

As mentioned earlier, labels and label selectors are among the more powerful *metadata management* features of Kubernetes. At its core, labels are a collection of text-based key/value pairs that can be attached to one or more objects: Pods, services, Deployments, and so on. Labels are intended to add information/semantics to objects that are relevant to the user and not the core Kubernetes system. A label selector is a method by which a user can group items together that have the same labels.

One of the most common uses of labels and label selectors in Kubernetes is the way that services use label selectors to group related Pods as endpoints for the service.

It's probably better shown by way of an example.

So, let's start with our three Infinispan Pods. Given that the Infinispan operator deploys its Pods via StatefulSets, the Pod names are pretty straightforward: `infinispan-0`, `infinispan-1`, `infinispan-2`. Take note of the labels attached to the Pods (highlighted in bold).

```
$ oc get pods --show-labels=true

NAME                                        READY   STATUS      RESTARTS   AGE
LABELS
infinispan-0                               1/1     Running     0          2m25s
app=infinispan-pod,clusterName=infinispan,controller-revision-hash=infinispan-
66785c8f,infinispan_cr=infinispan,statefulset.kubernetes.io/pod-
name=infinispan-0

infinispan-1                               1/1     Running     0          5m51s
app=infinispan-pod,clusterName=infinispan,controller-revision-hash=infinispan-
66785c8f,infinispan_cr=infinispan,statefulset.kubernetes.io/pod-
name=infinispan-1

infinispan-2                               1/1     Running     0          4m12s
app=infinispan-pod,clusterName=infinispan,controller-revision-hash=infinispan-
66785c8f,infinispan_cr=infinispan,statefulset.kubernetes.io/pod-
name=infinispan-2
```

When the Tournament service wants to connect to one of these Infinispan pods, it uses the Infinispan service that is also created and managed by the operator.

```
$ oc get svc

NAME         TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)      AGE
infinispan   ClusterIP   172.30.154.122   <none>     11222/TCP    5d20h
```

If we go into the definition of the service, we'll see the selector (highlighted in bold):

```
$ oc describe service infinispan
Name: infinispan
Namespace: pet-battle-tournament
Labels: app=infinispan-service
        clusterName=infinispan
        infinispan_cr=infinispan
Annotations: service.alpha.openshift.io/serving-cert-signed-by:
               openshift-service-serving-signer@1607294893
             service.beta.openshift.io/serving-cert-secret-name:
               infinispan-cert-secret
             service.beta.openshift.io/serving-cert-signed-by:
               openshift-service-serving-signer@1607294893
Selector:          app=infinispan-pod,clusterName=infinispan

Type:              ClusterIP
IP:                172.30.154.122
Port:              infinispan  11222/TCP
TargetPort:        11222/TCP
Endpoints: 10.128.2.158:11222,10.129.3.145:11222,10.131.0.25:11222
Session Affinity:  None
Events:            <none>
```

This adds the Pods with the labels `app=infinispan-pod,clusterName=infinispan` into the service as endpoints. Two things to note here: the selector didn't use all the labels assigned to the Pod; and if we scaled up the number of Infinispan Pods, the selector would be continuously assessed and the new Pods automatically added to the service. The preceding example is a pretty basic example of a selector; in fact, selectors are far more powerful, with equality- and set-based operations also available. Check out the examples in the Kubernetes documentation for more information.[11]

11 https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/

Great, so now what? What information could you use to label a resource? It depends on what your needs are. As demonstrated previously in the monitoring section, labels and selectors can be useful in configuring Prometheus. Labels can also be useful in grouping components together, as in the components that comprise a distributed application.

Kubernetes has a set of recommended labels[12] that we've used when building and deploying the PetBattle application:

Label	Description	Value
app.kubernetes.io/name	The name of the application	`pet-battle-tournament-service`
app.kubernetes.io/instance	A unique name identifying the instance of an application	`petbattle`
app.kubernetes.io/version	The current version of the application	`1.0.0`
app.kubernetes.io/component	The component within the architecture	`pet-battle-tournament-service`
app.kubernetes.io/part-of	The name of a higher-level application this one is part of	`petbattleworld`
app.kubernetes.io/managed-by	The tool being used to manage the operation of an application	`Helm`

Table 16.1: Kubernetes-recommended labels

12 https://kubernetes.io/docs/concepts/overview/working-with-objects/common-labels/

With these labels in place, it is possible to retrieve and view the components of the application using selectors, such as to show the component parts of the PetBattle application without the supporting application infrastructure, that is, Infinispan or Keycloak. The following command demonstrates this:

```
$ oc get all -l app.kubernetes.io/part-of=petbattleworld \
    --server-print=false

NAME                                                        AGE
replicationcontroller/dabook-mongodb-1                      2d18h
replicationcontroller/dabook-pet-battle-tournament-1        26m

NAME                                                        AGE
service/dabook-mongodb                                      2d18h
service/dabook-pet-battle-tournament                        2d18h

NAME                                                        AGE
deploymentconfig.apps.openshift.io/dabook-mongodb           2d18h
deploymentconfig.apps.openshift.io/dabook-pet-battle-tournament  26m

NAME                                                        AGE
imagestream.image.openshift.io/dabook-pet-battle-tournament 26m

NAME                                                        AGE
route.route.openshift.io/dabook-pet-battle-tournament       2d18h
```

Let's look at other mechanisms we can use to enhance traceability.

Software Traceability

One of the issues that we've observed from customers over the years is the reliance that people have on the name of the software artifact that they're putting into production, such as `super-important-app-1.2.99.0.bin or critical-service-1.2.jar`. While this works 99.9% of the time, occasionally we've noticed issues where an incorrect version has been deployed with interesting outcomes.

In the land of containers, your deployment is a versioned artifact that contains a version of your software, and this in turn may be deployed using a versioned Helm chart via a GitOps approach. A good build and deployment pipeline will ensure that these levels of artifact versioning will always be consistent and provide traceability. As a backup, we also add additional traceability to the deployed artifacts as annotations on the resources and build info logging in the application binary.

Annotations

Annotations are similar to Kubernetes labels—that is, string-based key/value pairs—except that they're not used to group or identify objects via selectors. Annotations can be used to store different types of information; in our case, we're going to use annotations to store Git information to help with software traceability.

```
apiVersion: v1
kind: Service
metadata:
  annotations:
    app.openshift.io/vcs-url:
      https://github.com/petbattle/tournamentservice.git
    app.quarkus.io/commit-id:a01a310aadd46911bc4c66b3a063ddb090a3feba
    app.quarkus.io/vcs-url:
      https://github.com/petbattle/tournamentservice.git
    app.quarkus.io/build-timestamp: 2020-12-23 - 16:43:07 +0000
    prometheus.io/scrape: "true"
    prometheus.io/path: /metrics
    prometheus.io/port: "8080"
```

The annotations are automatically added as part of the Maven build process using the Quarkus Maven plugin. Also notice the annotations are used to provide scrape information for Prometheus, as can be seen highlighted in the preceding code.

Build Information

An approach that has nothing to do with Kubernetes per se, but we strongly recommend to be used in general, is to output source control and build information as part of the application startup. An example of this is embedded into the Tournament service.

```
$ java -jar tournament-1.0.0-SNAPSHOT-runner.jar

GITINFO -> git.tags:
GITINFO -> git.build.version:1.0.0-SNAPSHOT
GITINFO -> git.commit.id.full:b5d6bfabeea6251b9c17ea52f0e87e2c8e967efd
GITINFO -> git.commit.id.abbrev:b5d6bfa
GITINFO -> git.branch:noc-git-info
GITINFO -> git.build.time:2020-12-24T13:26:25+0000
GITINFO -> git.commit.message.full:Moved gitinfo output to Main class
GITINFO -> git.remote.origin.url:git@github.com:petbattle/tournamentservice.
git
```

We use the Maven plugin `git-commit-id-plugin` to generate a file containing the Git information and package that file as part of the **Java archive (jar)**. On startup, we simply read this file and output its contents to the console. Very simple stuff, but very effective and a lifesaver when needed. When running on OpenShift, this information will be picked up by the OpenShift logging components.

Alerting

So we have all the metrics to provide us with some insight into how the system is performing. We've got spectacular graphs and gauges in Grafana but we're hardly going to sit watching them all day to see if something happens. It's time to add alerting to the solution.

What Is an Alert?

An alert is an event that is generated when some measurement threshold (observed or calculated) is about to be or has been breached. The following are some examples of alerts:

- The average system response time in the last five minutes goes above 100 milliseconds.
- The number of currently active users on the site falls below a certain threshold.
- Application memory usage is approaching its maximum limit.

Alerts usually result in notifications being sent to human operators, whether that is through an email or instant message, say. Notifications can also be sent to trigger automation scripts/processes to deal with the alert. Service owners can analyze their existing alerts to help improve the reliability of their services and systems and reduce the manual work associated with remediating problems.

Why Alert?

Alerts call for human action when a situation has arisen within the system that cannot be automatically handled. This may include scenarios where automatic resolution of the problem is deemed too risky and human intervention is required to help triage, mitigate, and resolve the issue. Alerting can also be an issue by causing concern for site reliability engineers who manage and operate the system, particularly when alerts are numerous, misleading, or don't really help in problem cause analysis. They may generate benign alerts that don't prompt any action.

There are certain qualities that make up a *good alert*. Alerts should be actionable by the human beings who respond to them. To be actionable, the alert must also have arrived in time for something to be done about it and it should be delivered to the correct team or location for triaging. Alerts can also include helpful metadata such as documentation links to assist in making triage faster.

Alert Types

We can think of alerts as falling into three broad categories.[13] The first are **proactive** alerts, meaning that your business service or system is not in danger yet but may be in trouble after some period of time. A good example of this is where your system response time is degrading but it is not at a stage where external users would be aware of the issue yet. Another example may be where your disk quota is filling up but is not 100% full yet, but it may do in a few days' time.

A **reactive** alert means your business service or system is in immediate danger. You are about to breach a service level and immediate action is needed to prevent the breach.

An **investigative** alert is one where your business service or system is in an unknown state. For example, it may be suffering a form of partial failure or there may be unusual errors being generated. Another example may be where an application is restarting too many times, which is indicative of an unusual crash situation.

Each of these alerts may also be directed to different teams, depending on their severity. Not all alerts need to be managed with the same level of urgency. For example, some alerts must be handled by an on-call human resource immediately, while for others it may be fine to handle them during business hours by an application business support team the following day. Let's explore how we can easily configure and add alerting to our applications using the OpenShift platform features to help us out.

13 https://www.oreilly.com/content/reduce-toil-through-better-alerting/

Managing Alerts

OpenShift has platform monitoring and alerting that supports both built-in platform components and user workloads. The product documentation is the best place to start when looking to configure these.[14] As we outlined earlier, monitoring and alerting make use of the Prometheus monitoring stack. This is combined with an open-source tool called Thanos[15] that aggregates and provides access to multiple instances of Prometheus in our cluster.

A basic configuration for the PetBattle application suite consists of creating two ConfigMaps for user workload monitoring and alerting. We use ArgoCD and a simple kustomize YAML configuration to apply these ConfigMaps using GitOps. If we open up the ubiquitous journey `values-day2ops.yaml` file, we can create an entry for user workload monitoring.

```
# User Workload Monitoring
- name: user-workload-monitoring
  enabled: true
  destination: openshift-monitoring
  source: https://github.com/rht-labs/refactored-adventure.git
  source_path: user-workload-monitoring/base
  source_ref: master
  sync_policy: *sync_policy_true
  no_helm: true
```

The next step is to make use of application metrics and a ServiceMonitor and configure specific Prometheus alerts for our PetBattle suite.

User-Defined Alerts

In the *Metrics* section, we created ServiceMonitors for our API and Tournament applications that allow us to collect the micrometer metrics from our Quarkus applications. We want to use these metrics to configure our alerts. The simplest approach is to browse to the Thanos query endpoint that aggregates all of our Prometheus metrics. You can find this in the `openshift-monitoring` project.

```
$ oc get route thanos-querier -n openshift-monitoring
```

14 https://docs.openshift.com/container-platform/4.7/monitoring/configuring-the-monitoring-stack.html#configuring-the-monitoring-stack

15 https://github.com/thanos-io/thanos

We want to create a simple reactive alert based on whether the PetBattle API, Tournament, and UI Pods are running in a certain project. We can make use of Kubernetes Pod labels and the Prometheus query language to test whether our Pods are running.

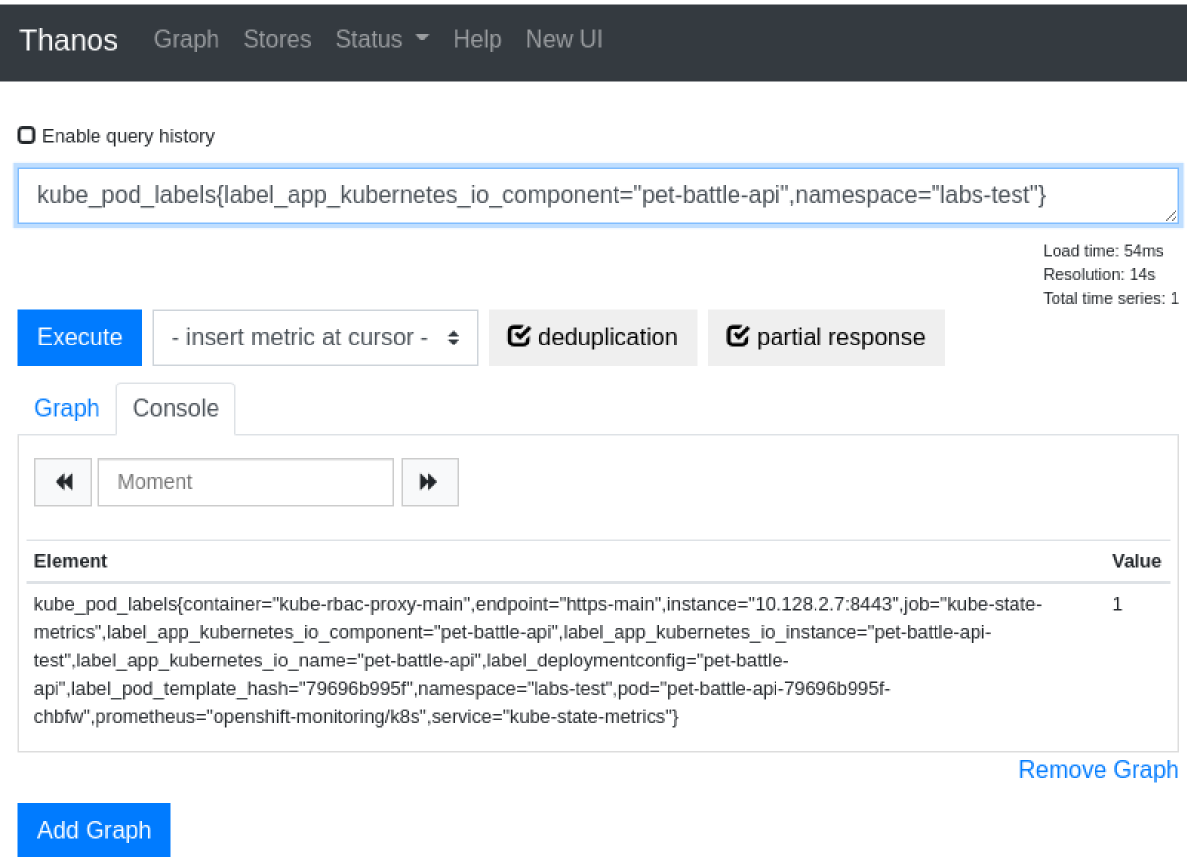


Figure 16.3: Thanos query interface

For this use case, we combine the `kube_pod_status_ready` and `kube_pod_labels` query for each Pod and namespace combination and create a PrometheusRule to alert us when a condition is not met. We wrapped the generation of the alerts in a Helm chart so we can easily template the project and alert severity values[16] and connect the deployment with our GitOps automation.

16 https://github.com/petbattle/ubiquitous-journey/blob/main/applications/alerting/chart/templates/application-alerts.yaml

```
  spec:
    groups:
    - name: petbattle.rules
      rules:
      - alert: PetBattleApiNotAvailable
        annotations:
          message: 'Pet Battle Api in namespace {{ .Release.Namespace }} is
  not available for the last 1 minutes.'
        expr: (1 - absent(kube_pod_status_ready{condition="true" ... for: 1m
        labels:
          severity: {{ .Values.petbattle.rules.severity }}
```

The firing alerts can be seen in the OpenShift web console as seen in *Figure 16.4*. In this example, we have configured the `labs-dev` alerts to only have a severity of *info* because they are not deemed as crucial deployments in that environment. The severity may be set as *info*, *warning*, or *critical*, and we use *warning* for our `labs-test` *and* `labs-staging` environments, for example. These are arbitrary but standard severity levels, and we can use them for routing alerts, which we will cover in a moment.

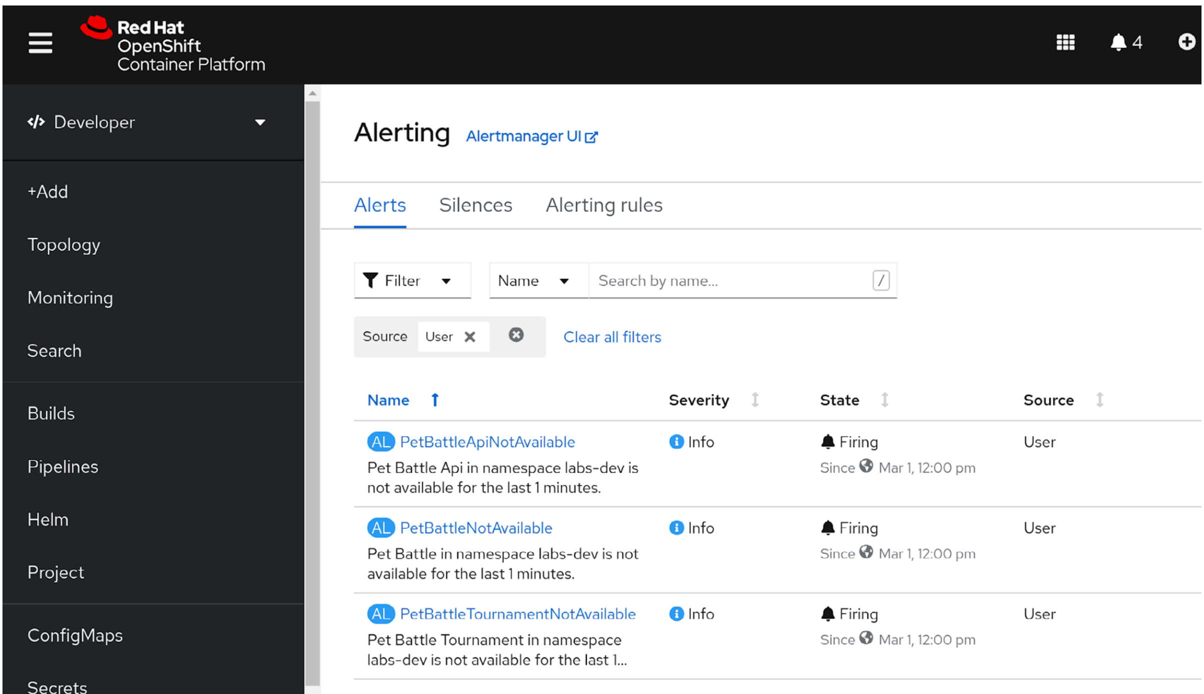


Figure 16.4: PetBattle alerts firing in OpenShift

We can use the same method to create an investigative or proactive alert. This time we wish to measure the HTTP request time for our API application. During testing, we found that if API calls took longer than ~1.5 sec, the user experience in the PetBattle frontend was deemed too slow by end users and there was a chance they would disengage from using the web application altogether.

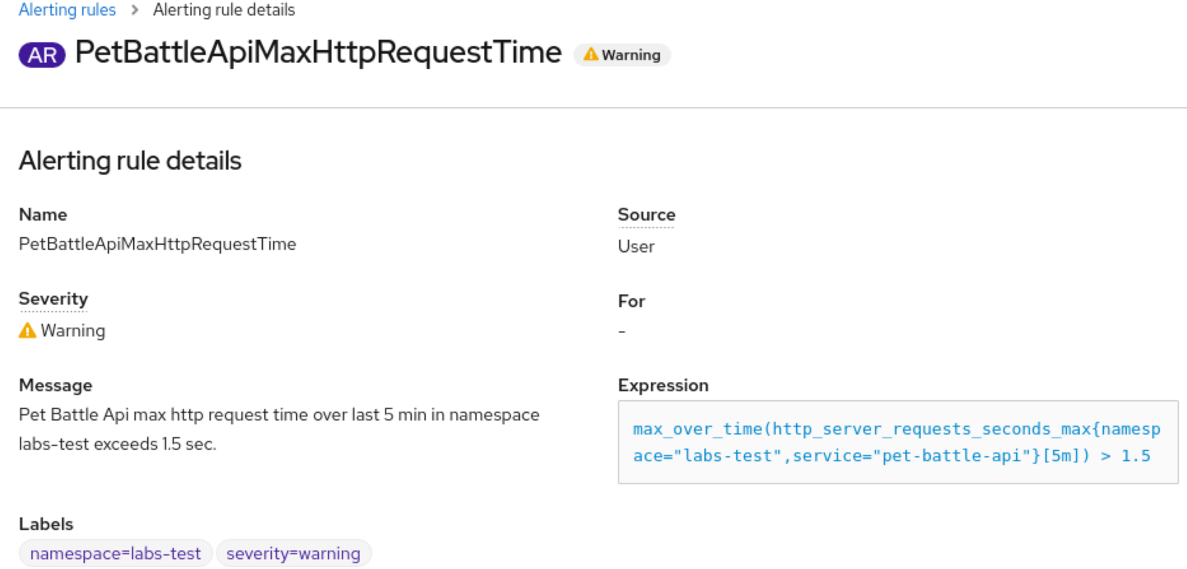


Figure 16.5: Maximum request time alert rule

In this alert, we use the Prometheus query language and the `http_server_requests_seconds_max` metric for the PetBattle API application to test whether the maximum request time over the last five-minute period exceeded our 1.5 sec threshold. If this alert starts to fire, possible remediation actions might include manually scaling up the number of API Pods or perhaps increasing the database resources if that is seen to be slow for some reason. In future iterations, we may even try to automate the application scale-up by using a Horizontal Pod Autoscaler, a Kubernetes construct that can scale our applications automatically based on metrics.

In this way, we can continue to build on our set of alerting rules for our PetBattle application suite, modifying them as we run the applications in different environments, and learn what conditions to look out for while automating as much of the remediation as we can.

OpenShift Alertmanager

As we have seen, OpenShift supports three severity levels of alerting: *info*, *warning*, and *critical*. We can group and route alerts based on their severity as well as on custom labels—that is, project or application labels. In the OpenShift administrator console,[17] you can configure the Alertmanager under **Cluster Settings**.

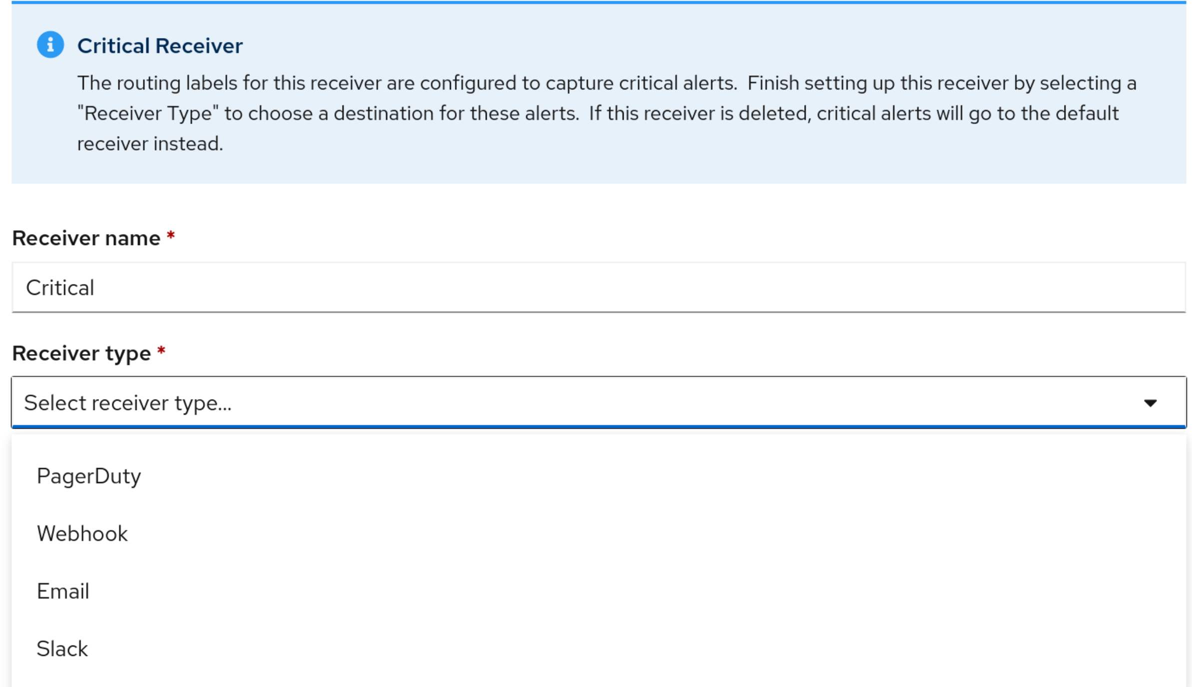


Figure 16.6: Alertmanager routing configuration

Alerts may be grouped and filtered using labels and then routed to specific receivers, such as PagerDuty, Webhook, Email, or Slack. We can fine-tune the routing rules so that the correct teams receive the alerts in the correct channel, based on their urgency. For example, all *info* and *warning* severity alerts for the PetBattle UI application may be routed to the *frontend developers* Slack channel, whereas all *critical* alerts are routed to the on-call PagerDuty endpoint as well as the Slack channel.

17 https://docs.openshift.com/container-platform/4.7/monitoring/managing-alerts.html

Alerting is a critical component to successfully manage the operational aspects of a system but you should be careful and ensure that the operations team isn't overwhelmed with alerts. Too many alerts or many minor or false-positive alerts can lead to *alert fatigue*, where it becomes an established practice within a team to ignore alerts, thus robbing them of their importance to the successful management of the system.

Service Mesh

Service mesh functionality has been one of the largest additions/extensions to Kubernetes in its short history. There's a lot of debate around the additional complexity of using a service mesh and whether all the features are even required.

For the purposes of this book, we're going to focus on the service mesh provided out of the box within OpenShift, which is based on the open-source Istio project. There are other implementations, such as Linkerd, SuperGloo, and Traefik, out there that are excellent and offer similar functionality to Istio.

The OpenShift service mesh provides the following features out of the box:

- **Security**: Authentication and authorization, mutual TLS (encryption), policies
- **Traffic management**: Resiliency features, virtual services, policies, fault injection
- **Observability**: Service metrics, call tracing, access logs

Why Service Mesh?

We previously talked about resiliency and how patterns like circuit breakers can help systems recover from downstream failures. A circuit breaker can be added in the scope of application code through frameworks such as **SmallRye Fault Tolerance** or **Spring Cloud Circuit Breaker** for Java projects; similar frameworks such as **Polly**[18] exist for .NET, **PyBreaker**[19] for Python, and **Opossum**[20] for Node.js. A key requirement for all of these frameworks is that they have to be added to the existing source code of the application, and the application needs to be rebuilt. When using a service mesh, a circuit breaker is external to the application code and no changes are required at the application level to take advantage of this feature.

18 https://github.com/App-vNext/Polly

19 https://pypi.org/project/pybreaker/

20 https://nodeshift.dev/opossum/

The same is true with **Mutual TLS** (**mTLS**), which is used for encrypting traffic between services. Operators such as CertManager or CertUtil can assist with managing and distributing certificates, but modification of the application code is still required to use the feature. Service meshes simplify this as the inter-component traffic is sent via a *sidecar proxy* and functionality such as mTLS is *automagically* added to this—once again, without having to change the application code.

The Istio component of a service mesh also manages TLS certificate generation and distribution so that it helps reduce the management overhead when using mTLS.

So how does a service mesh perform all of this magical functionality? Basically, the service mesh operator adds a service proxy container (based on the Envoy project) to the application Pod and configures the application traffic to be routed through this proxy. The proxy registers with the Istio control plane and configuration settings, certificates, and routing rules are retrieved and the proxy configured. The Istio documentation goes into much more detail.[21]

Aside – Sidecar Containers

A common object that people visualize when they hear the word *sidecar* is that of a motorbike with a single-wheel passenger car—a *pod*—attached to it. Being attached to the bike, the pod goes wherever the bike goes—except in comedy sketches where the bike and sidecar separate and rejoin, but that's another subject entirely.

In Kubernetes, a sidecar is a container that runs in the same Kubernetes Pod as the main application container. The containers share the same network and ICP namespace and can also share storage. In OpenShift, when using the service mesh functionality, a Pod annotated with the correct annotation `sidecar.istio.io/inject: "true"` will have an Istio proxy automatically injected as a sidecar alongside the application container. All subsequent communications between the application and external resources will flow through this sidecar proxy and hence enable the usage of features such as circuit breakers, tracing, and TLS, as and when they are needed. As the great Freddie Mercury once said, "*It's a kind of magic*."

```
# Let's patch the deployment for our pet battle apps
# running in petbattle ns and istioify it
$ helm upgrade \
--install pet-battle-tournament \
--version=1.0.39 \
--set pet-battle-infra.install_cert_util=true \
--set istio.enabled=true \
--timeout=10m \
```

21 https://istio.io/latest/docs/

```
--namespace petbattle
petbattle/pet-battle-tournament
$ oc get deployment pet-battle-tournament -o yaml \
--namespace petbattle
...
template:
    metadata:
      annotations:
...
        sidecar.istio.io/inject: "true"
      labels:
        app.kubernetes.io/component: pet-battle-tournament
        app.kubernetes.io/instance: pet-battle-tournament
        app.kubernetes.io/name: pet-battle-tournament
```

It is possible to have more than one sidecar container if required. Each container can bring different features to the application Pod: for example, one for Istio, another for log forwarding, another for the retrieval of security credentials, and so on. It's easy to know when a Pod is running more than a single container; for example, the READY column indicates how many containers are available per Pod and how many are ready—that is, its readiness probe has passed.

```
$ oc get pods

NAME                                READY   STATUS     RESTARTS   AGE
infinispan-0                       1/1     Running    0          4h13m
Pet-battle-3-68fm5                  2/2     Running    0          167m
pet-battle-api-574f77ddc5-l5qx8 2/2     Running    0          163m
pet-battle-api-mongodb-1-7pgfd  1/1     Running    0          3h50m
pet-battle-tournament-3-wjr6r   2/2     Running    0          167m
pet-battle-tou.-mongodb-1-x7t9h 1/1     Running    0          4h9m
```

Be aware, though, that there is a temptation to try and utilize all the service mesh features at once, known as the *ooh… shiny* problem.

Here Be Dragons!

The adoption of a service mesh isn't a trivial exercise when it comes to complex solutions with multiple components and development teams. One thing to understand about a service mesh is that it crosses a lot of team boundaries and responsibilities. It includes features that are focused on the developer, operations, and security teams; all of these teams/personnel need to work together to understand and get the best out of using the features provided by the mesh. If you're just starting out, our advice is to start small and figure out what features are necessary in production and iterate from there.

In the case of PetBattle, we decided that we were going to primarily focus on using some of the features in the areas of traffic management and observability. The rationale behind this was that Keycloak already addressed many of the security requirements, and we also wanted to finish the book before the end of the decade.

Service Mesh Components

The functionality of the service mesh is made up of a number of independent components:

- Jaeger and Elasticsearch provide the call tracing functionality and logging functionality.
- Kiali provides the mesh visualization functionality.
- OpenShift Service Mesh provides the core Istio functionality.

The good news is that all of these components are installed and managed by Operators, so installation is reasonably straightforward. These components are installed via Helm, and if you want to know more about how they are installed, then the Red Hat OpenShift documentation will have the relevant details.

One key thing to note is that at the time of writing this book, OpenShift Service Mesh ships with a downstream version of Istio called Maistra. This is primarily due to the out-of-the-box multi-tenancy nature of OpenShift, as well as limiting the scope of Istio cluster-scoped resources. OpenShift Service Mesh also ships with an **Istio OpenShift Routing** (**IOR**) component that maps the Istio gateway definitions onto OpenShift routes. Note that Istio is still the upstream project and bugs/feature requests are fixed/implemented, as necessary.

For traffic management, Istio has the following core set of resources:

- **Gateways**: Controls how traffic gets into the mesh from the outside, akin to OpenShift routes.
- **Virtual service**: Controls how traffic is routed within the service mesh to a destination service. This is where functionality such as timeouts, context-based routing, retries, mirroring, and so on, are configured.
- **Destination rule**: Service location where traffic is routed to once traffic rules have been applied. Destination rules can be configured to control traffic aspects such as load balancing strategies, connection pools, TLS setting, and outlier detection (circuit breakers).

There are other resources such as service entry, filters, and workloads, but we're not going to cover them here.

PetBattle Service Mesh Resources

We'll briefly introduce some of the resources that we use in PetBattle and explain how we use them.

Gateways

The gateway resource, as stated earlier, is used to create an ingress route for traffic coming into the service mesh.

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: petbattle-gateway-tls
spec:
  selector:
    istio: ingressgateway
  servers:
  - port:
      number: 443
      name: https
      protocol: HTTPS
    tls:
      mode: SIMPLE
      credentialName: "pb-ingressgateway-certs"
    hosts:
    - "*"
```

A few things to note with this definition is that it will create an OpenShift route in the *istio-system* namespace and not the local namespace. Secondly, the route itself will use SSL, but it won't be able to utilize the OpenShift router certificates by default. Service mesh routes have to provide their own certificates. As part of writing this book, we took the pragmatic approach and copied the OpenShift router certificates into the *istio-system* namespace and provided them to the gateway via the *pb-ingressgateway-certs* secret. Note that this is for demonstration purposes only—*do not try this in production*. The correct approach for production is to generate and manage the PKI using as-a-service certificates.

Virtual services

PetBattle contains a number of VirtualServices, such as *pet-battle-cats-tls*, *pet-battle-main-tls*, and *pet-battle-tournament-tls*.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: pet-battle-cats-tls
spec:
  hosts:
  - "*"
  gateways:
  - petbattle-gateway-tls
  http:
  - match:
    - uri:
        prefix: /cats
      ignoreUriCase: true
    route:
    - destination:
        host: pet-battle-api
        port:
          number: 8080
    retries:
      attempts: 3
      perTryTimeout: 2s
      retryOn: gateway-error,connect-failure,refused-stream
```

The VirtualServices are all similar in function in that they are all configured to:

1. Match a specific URI; in the example above, `/cats`.
2. Once matched, route the traffic to a specific destination.
3. Handle specific errors by performing a fixed number of request retries.

Destination Rule

Finally, the traffic is sent to a destination or even distributed to a set of destinations depending on the configuration. This is where DestinationRules come into play.

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: pet-battle-api-port
spec:
  host: pet-battle-api.prod.svc.cluster.local
  trafficPolicy: # Apply to all ports
    portLevelSettings:
    - port:
        number: 8080
      loadBalancer:
        simple: LEAST_CONN
```

In our example, the traffic sent to a specific port is load balanced based on a simple strategy that selects the Pod with the least number of active requests. There are many load balancing strategies that can be used here, depending on the needs of the application—everything from simple round robin to advanced consistent hashing load balancing strategies, which can be used for session affinity. As ever, the documentation goes into far greater detail.[22]

We can visualize the flow of traffic from the above example, as seen in *Figure 16.7*:

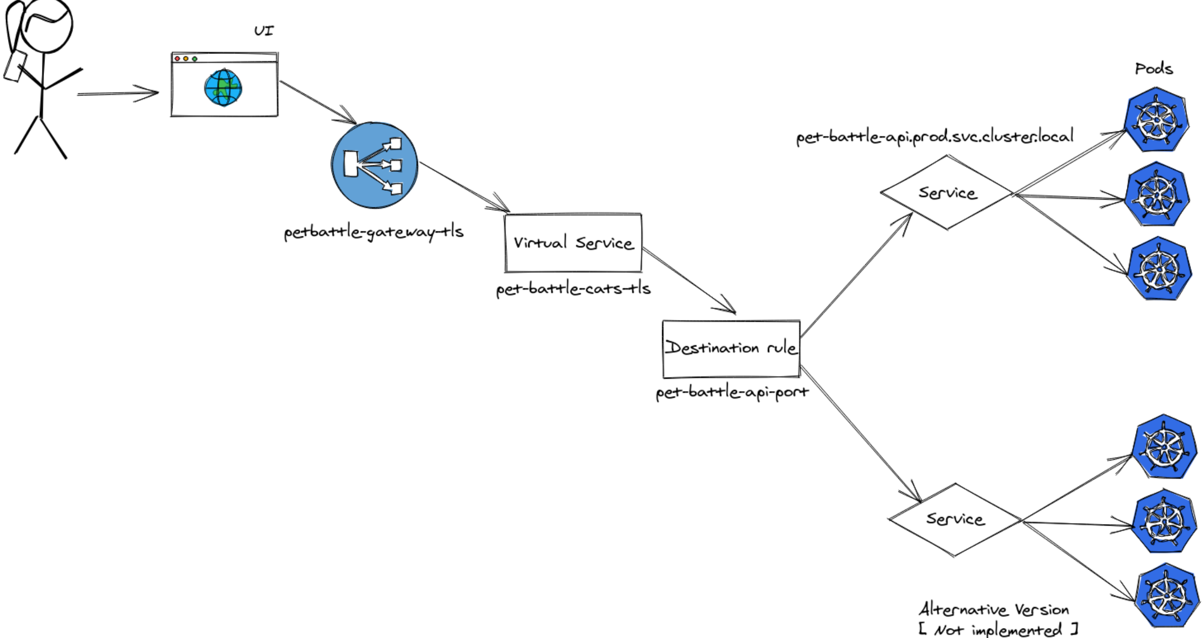


Figure 16.7: PetBattle traffic flow

22 https://istio.io/latest/docs/

Note that *Figure* 16.7 shows an example of how destination rules can be used to send traffic to an alternative version of the service. This can be useful for advanced deployment strategies such as Canary, Blue/Green, and so on. We haven't discussed how to do this with OpenShift Service Mesh in this book, but the reader is encouraged to explore this area in more detail. A good place to start is the aforementioned Istio documentation.

Managing all of these resources is reasonably simple when it's just a few services, and PetBattle utilizes service mesh functionality in a very basic manner. However, when there are many services and features, such as multiple destinations used in advanced deployment models, the amount of settings and YAML to interpret can be overwhelming. This is where mesh visualization functionality can be useful to visualize how all of this works together. For this, we use the Kiali functionality, which is part of OpenShift Service Mesh. *Figure* 16.8 shows how PetBattle is visualized using Kiali.

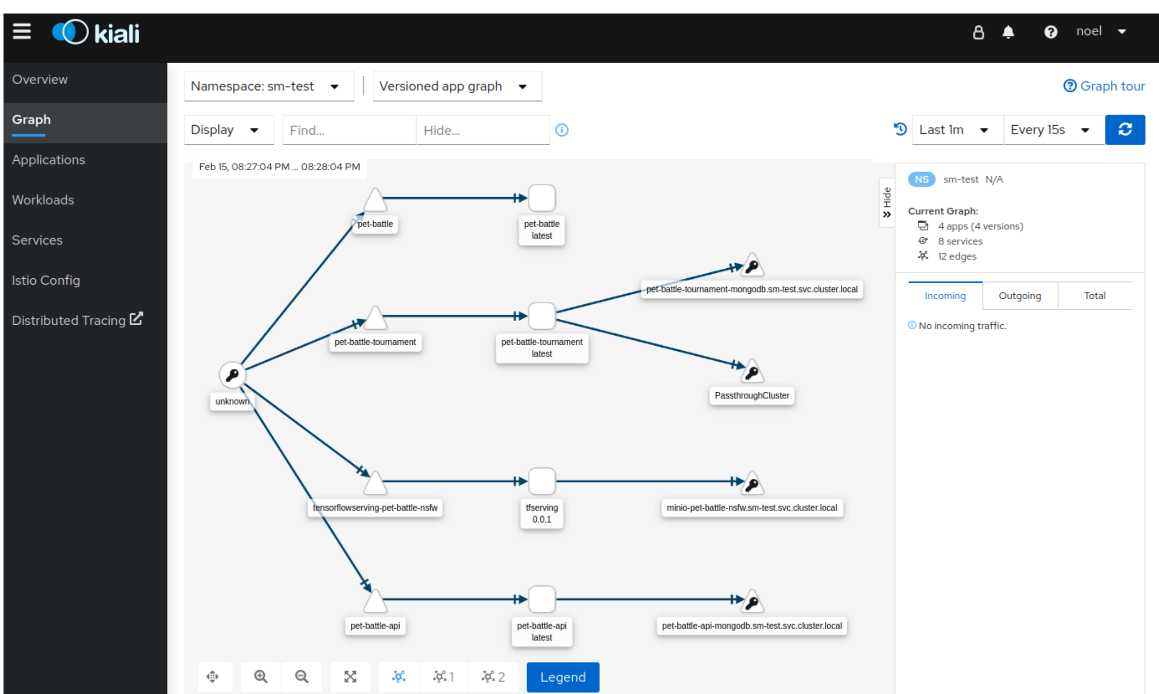


Figure 16.8: Kiali service graph for PetBattle

Kiali can be very useful for diagnosing the current state of the mesh, as it can dynamically show where traffic is being sent as well as the state of any circuit breakers being used. It also integrates with Jaeger for tracing requests across multiple systems. Kiali can also help prevent configuration issues by semantically validating the deployed service mesh resources.

Next we're going to explore one of the most powerful features of OpenShift 4 - Operators.

Operators Everywhere

Fundamental to the OpenShift 4 platform is the concept of *Operators*. So far, we have used them without talking about why we need them and what they actually represent on a Kubernetes platform such as OpenShift. Let's cover this briefly without totally rewriting the book on the subject.[23]

At its heart, the Operator is a software pattern that codifies knowledge about the running and operation of a particular software application. That application could be a distributed key value store, such as etcd. It might be a web application such as the OpenShift web console. Fundamentally, the operator can represent *any* application domain that could be codified. A good analogy for an operator is the *expert system*, a rules-based bit of software that represents knowledge about a certain thing that is put to work in a meaningful way. If we take a database as an example, the Operator might codify what a real human database administrator does on a day-to-day basis, such as the deployment, running, scaling, backup, patching, and upgrading of that database.

The physical runtime for an operator is nothing more than a Kubernetes Pod, that is, a collection of containers that run on a Kubernetes platform such as OpenShift. Operators work by extending or adding new APIs to the existing Kubernetes and OpenShift platform APIs. This new endpoint is called a **Custom Resource** (**CR**). CRs are one of the many extension mechanisms in Kubernetes.

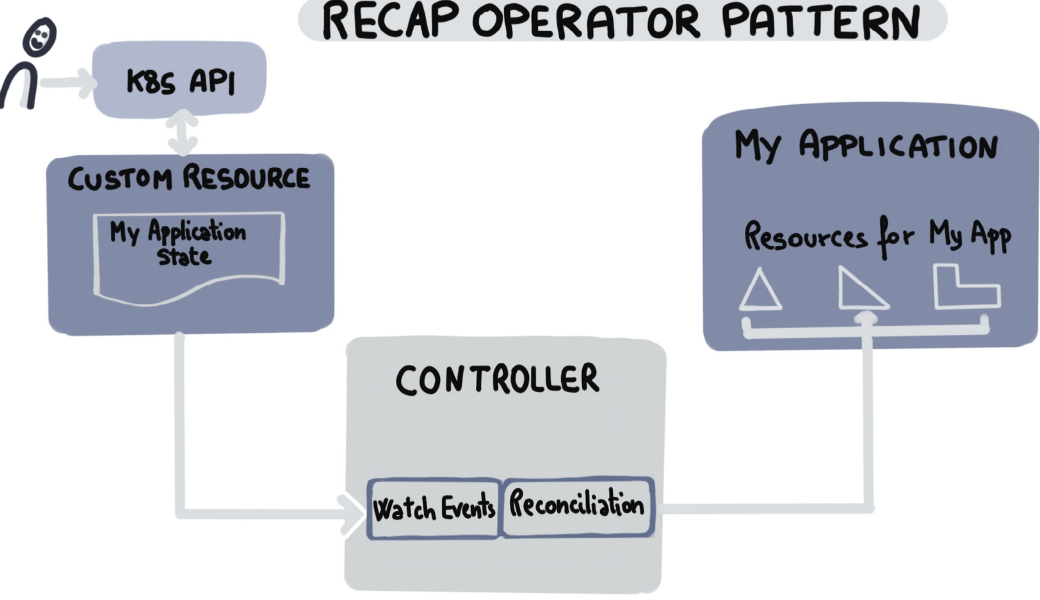


Figure 16.9: The Operator pattern

23 https://www.redhat.com/en/resources/oreilly-kubernetes-operators-automation-ebook

A **Custom Resource Definition** (**CRD**) defines what the CR is. Think of it as the definition or schema for the CR. The Operator Pod *watches* for events on the platform that are related to their custom resources and takes *reconciliation* actions to achieve the desired state of the system. When an Operator Pod stops or is deleted from the cluster, the application(s) that it manages should continue to function. Removing a CRD from your cluster does affect the application(s) that it manages. In fact, deleting a CRD will in turn delete its CR instances. This is the Operator pattern.

With Operators, all of the operational experience required to run/manage a piece of software can be packaged up and delivered as a set of containers and associated resources. In fact, the whole of the OpenShift 4 platform exists as a collection of operators! So, as the platform owner, you are receiving the most advanced administrator knowledge bundled up through Operators. Even better, Operators can become more advanced over time as new features and capabilities are added to them. A good understanding of how to configure Operators is required for OpenShift platform administrators. This usually involves setting properties in the OpenShift cluster global configuration web console, setting CR property values, using ConfigMaps, or similar approaches. The product documentation[24] is usually the best place to find out what these settings are for each Operator.

Figure 16.10: OpenShift Cluster Settings—configuring platform Operators

24 https://docs.openshift.com/container-platform/4.7

In OpenShift, the lifecycle management (upgrading, patching, managing) of Operators themselves is automated through the **Operator Lifecycle Manager** (**OLM**). These components make upgrading the OpenShift platform itself a lot more reliable and easier to manage from a user's perspective; it massively reduces the operational burden. Because Operators themselves are delivered as versioned images, we get the same benefits from immutable container images that we do for our own applications, i.e., the same image version can be run consistently in multiple cloud environments increasing quality and eliminating snowflakes (unique applications for unique environments).

And it is not just the OpenShift platform itself that can take advantage of Operators. The sharing and distribution of software using Operators via the Operator Hub[25] is open to software developers and vendors from all over the world. We use OLM and Operator *subscriptions* to deploy these into our cluster. The tooling required to build and develop Operators (the SDK) is open source and available to all.[26]

So, should *all* applications be delivered as Operators? The short answer is no. The effort required to code, package, test, and maintain an Operator may be seen as overkill for many applications. For example, if your applications are not being distributed and shared with others, and you only need to build, package, deploy, and configure your application in a few clusters, there are many simpler ways this can be achieved, such as using container images, Kubernetes, and OpenShift native constructs (BuildConfigs, Deployments, ReplicaSets, ConfigMaps, Secrets, and more) with tools such as Helm to achieve your goals.

Operators Under the Hood

To fully understand how operators work, you need to understand how the Kubernetes control loop works.

Control Loops

In very basic terms, core Kubernetes is just a **Key-Value** (**KV**) store—an etcd datastore with an API. Processes use this API to perform **Create**, **Read**, **Update**, and **Delete** (**CRUD**) actions on keys within the KV store. Processes can also register with the KV store to be notified when there are value changes to keys or sets of keys that they're interested in.

25 https://operatorhub.io

26 https://github.com/operator-framework/operator-sdk

When these processes get a change notification, they react to that notification by performing some activity, such as configuring iptables rules, provisioning storage, and so on. These processes understand the current state of the system and the desired state and work toward achieving that desired state. In other words, these processes are performing the role of a *control loop*, meaning they attempt to bring the state of the system to a desired state from where it currently resides.

In this example, the *process* is a controller that observes the state of a resource or set of resources and then makes changes to move the resource state closer to the desired state. As consumers of Kubernetes, we constantly use controllers. For example, when we instruct Kubernetes to deploy a Pod, the Pod controller works to make that a reality. Control loops are key to the operation on Kubernetes and it's a declarative and, eventually, consistent approach. For much more information, take a look at the Kubernetes Controller docs[27] and the OpenShift blog site for recommendations on how to build your own Operator.[28]

Operator Scopes

Operators can either be cluster-scoped or namespace-scoped. A cluster-scoped operator is installed once in a namespace and can create and manage resources in other namespaces; that is, cluster-wide. The OpenShift service mesh operator and its related operators such as Kiali and Jaeger are cluster-scoped. They are installed by default into the `openshift-operators` or `openshift-operators-redhat` namespace and create and manage resources when a related CRD is deployed in another namespace, such as PetBattle.

A namespace-scoped operator is one that is deployed in a namespace and only manages resources in that namespace. We use a number of these in PetBattle, such as Cert-Utils and Keycloak.

All Operators are installed via a CRD called a **Subscription**. Without going into too much detail (see the official documentation for more), a Subscription describes how to retrieve and install an instance of an operator. The following is an example of a Subscription that we use to install the Grafana operator.

27 https://kubernetes.io/docs/concepts/architecture/controller/

28 https://www.openshift.com/blog/kubernetes-operators-best-practices

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
 name: grafana-operator
spec:
 channel: alpha
 installPlanApproval: Automatic
 name: grafana-operator
 source: community-operators
 sourceNamespace: openshift-marketplace
 startingCSV: grafana-operator.v3.7.0
```

To see some of the namespace-scoped operators that PetBattle needs, run the following command.

```
$ oc get subscriptions

NAME                 PACKAGE              SOURCE                CHANNEL
cert-utils-operator cert-utils-operator community-operators   alpha
grafana-operator    grafana-operator    community-operators   alpha
infinispan         infinispan          community-operators   2.1.x
keycloak-operator   keycloak-operator   community-operators   alpha
```

Let us now take a look at how operators can be used by our PetBattle team.

Operators in PetBattle

We use operators to create and manage resources such as Infinispan cache and Keycloak SSO instances. We simply install the Infinispan operator and deploy a relevant custom resource to tell it to create and manage a replicated cache. We don't have to know about spinning up Infinispan Pods or creating SSL certificates or provisioning storage space. The operator will do all of this for us, and if something fails or is accidentally deleted, the operator will look after the recreation of the resource. In the Infinispan example, if we delete the Infinispan K8s service, the operator will be notified about its deletion and recreate the service automatically. As developers, we don't have to worry about managing it.

It is simpler to think of Operators as *looking after stuff so you don't have to*. It is also possible to use multiple operators in combination to automate complex workflows. For example, we use Keycloak for its SSO gateway and user management functionality. The Keycloak instance is deployed and managed via a Keycloak Operator. We just need to build and send a custom resource to the API and the operator will do the rest. One of the resources managed by the Operator is a Kubernetes Secret that contains the TLS certificates and keys, which clients interacting with the Keycloak instance will need to use. Given that Keycloak is the security gateway to our application, it is prudent to ensure that all communications are encrypted. However, this causes issues for Java-based applications; to use SSL, the JVM requires that it be provided with a Java TrustStore containing the SSL/TLS certificates and keys so that the JVM can trust them.

So, how do we take the Secret with the TLS certificates and keys and convert that into a TrustStore that the Java applications can use? We could do a whole heap of scripting with Bash, the Java Keytool, and potentially other tools to extract the certs/keys, creating the TrustStore, converting, and finally injecting the certs/keys into said TrustStore. This is manual, complex, and error-prone work. We will also have to recreate these TrustStores for each environment and handle lifecycle events such as certificate expiry.

Alternatively, we could use an operator, in this case, the *Cert-Utils* operator. We first install the *Cert-Utils* operator in the PetBattle namespace. This Operator was developed by the Red Hat Consulting PAAS Community of Practice[29] to help manage certificates and JVM Keystores, along with TrustStores.

To use this Operator, we first create a ConfigMap containing a set of specific annotations. The Cert-Utils operator will detect these annotations and create a TrustStore containing the relevant certificates and keys; it will also add the TrustStore to the ConfigMap. Finally, we can mount the ConfigMap into a Deployment and instruct the JVM to use that TrustStore. The following resource definition will create the TrustStore with the relevant certificates and keys.

```
apiVersion: v1
kind: ConfigMap
metadata:
  annotations:
    service.beta.openshift.io/inject-cabundle : "true"
    cert-utils-operator.redhat-cop.io/generate-java-truststore: "true"
    cert-utils-operator.redhat-cop.io/source-ca-key: "service-ca.crt"
    cert-utils-operator.redhat-cop.io/java-keystore-password: "jkspassword"
  name: java-truststore
```

This does the following:

29 https://github.com/redhat-cop/cert-utils-operator

- The `service.beta.openshift.io/inject-cabundle` annotation will inject the service signing certificate bundle into the ConfigMap as a `service-sa.crt` field.
- The `cert-utils-operator.redhat-cop.io` annotation will create the Java TrustStore in the ConfigMap under the name `truststore.jks` with the `jkpassword` password.

In the Tournament service, the following Quarkus configuration will mount the `java-truststore` ConfigMap and configure the JVM accordingly.

```
# Mount the configmap into the application pod in the /tmp/config/ directory
quarkus.kubernetes-config.enabled=true
quarkus.openshift.config-map-volumes.javatruststore.config-map-name=java-
truststore
quarkus.openshift.mounts.javatruststore.path=/tmp/config/

# Instruct the JVM to use the Truststore
quarkus.openshift.env-vars.JAVA_OPTS.value=-Djavax.net.ssl.trustStore=/tmp/
config/truststore.jks -Djavax.net.ssl.trustStorePassword=jkspassword

# Tell Infinispan client to use the Truststore when connecting
quarkus.infinispan-client.trust-store=/tmp/config/truststore.jks
quarkus.infinispan-client.trust-store-password=jkspassword
```

We've only just scratched the surface of operators. OpenShift ships with a number of supported operators and there are many community operators available as well. We used many community-based operators in this book, such as the Infinispan operator and Keycloak operator; there are productized versions of these operators available as well. There are many more operators from multiple vendors available from OperatorHub.[30]

It is also possible to write your own operators if required. The OperatorFramework[31] is an open-source SDK with which you can write your own operators using either Go, Ansible, or Helm.

30 https://operatorhub.io/

31 https://operatorframework.io/

Service Serving Certificate Secrets

Keycloak uses an OpenShift feature called *service serving certificate secrets*.[32] This is used for traffic encryption. Using this feature, OpenShift automatically generates certificates signed by the OpenShift certificate authority and stores them in a secret. The application, in this case Keycloak, can then mount this secret and use these certificates to encrypt traffic. Any application interacting with a Keycloak instance then just has to trust these certificates. OpenShift also manages the lifecycle of these certificates and automatically generates new certificates when the existing certificates are about to expire.

To turn on this feature, simply add the following annotation to a service:

```
service.beta.openshift.io/serving-cert-secret-name=<NameOfMysecret>
```

In the case of Keycloak, the operator does this as part of its processing:

```
$ oc get svc keycloak -o yaml

apiVersion: v1
kind: Service
metadata:
  annotations:
    description: The web server's https port.
    service.alpha.openshift.io/serving-cert-secret-name: sso-x509-https-
secret
    service.alpha.openshift.io/serving-cert-signed-by: openshift-service-
serving-signer@1615684126
    service.beta.openshift.io/serving-cert-signed-by: openshift-service-
serving-signer@1615684126
```

The secret contains the actual certificate and associated key:

```
$ oc get secret sso-x509-https-secret -o yaml

apiVersion: v1
data:
  tls.crt: …...
  tls.key: …..
kind: Secret
metadata:
  annotations:
```

32 https://docs.openshift.com/container-platform/4.7/security/certificates/service-serving-certificate.html

And it contains the certificate details as well:

```
$ oc get secret sso-x509-https-secret -o json \
    | jq -r '.data."tls.crt"' | base64 --decode \
    |openssl x509  -text -noout

Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 1283774295358672234 (0x11d0e1eb7ea18d6a)
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: CN = openshift-service-serving-signer@1615684126
        Validity
            Not Before: Mar 15 08:34:54 2021 GMT
            Not After : Mar 15 08:34:55 2023 GMT
        Subject: CN = keycloak.labs-staging.svc
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
```

Such Operator patterns simplify the burden of running complex middleware infrastructure applications on the OpenShift platform.

Conclusion

To be able to successfully run your software at scale in production, a good understanding of the instrumentation that surrounds the software stack is required. OpenShift is a modern platform that provides all of the capabilities required to observe and, in a lot of cases, automatically heal your applications while they're running.

In this chapter, we have discussed many common technical patterns that allow application developers to make use of these common platform capabilities. For example, one of the simplest patterns is to always log to STDOUT so the platform logging mechanisms can be leveraged. With containers, it becomes an antipattern to log to specific files mounted in a temporary filesystem within your container, because they are not clearly visible.

More complex patterns are also important to keep your business service applications running, even during disruption and change. Correctly configuring liveness, readiness, and startup probes so that your application can deploy without loss of service, configuring Pod disruption budgets for when nodes are restarted. Using application features to expose Prometheus metric endpoints for alerting and monitoring on the platform is a great way to alert teams when human interaction is required.

The service mesh is an advanced extension to OpenShift, extrapolating many features that would have traditionally been packaged into your applications so they can be more efficiently managed at the platform level. This is a common theme: taking application and development cross-cutting features and leveraging them to the benefit of all platform services.

The Operator pattern eases the operational burden of running complex middleware infrastructure applications on the OpenShift platform, packaging all the years of expert knowledge as software. It is no secret that OpenShift itself uses this fantastic pattern for all of its core capabilities. The real power comes in being able to lifecycle manage this complexity in an automated manner. Human toil is massively reduced because the system can self-heal and auto-upgrade without interference. Doing more with less is still the name of the game.

As a cross-functional product team, once you have learned and mastered these capabilities, it really does become possible to *give the developers the pagers*. The quality of any business service delivery starts with business discovery, which then transitions to application software, expands through to platform capabilities, and finally on and out into the world of networking and end user devices connected via the internet. Once developers and cross-functional product teams are empowered to build, run, and own their software—in every environment that it is required to run in—only then will they fully equate and connect happy customers with the software supply chain that they code, automate, and continuously deliver.

Praise for DevOps Culture and Practice with OpenShift

"Creating successful, high-performing teams is no easy feat. DevOps Culture and Practice with OpenShift provides a step-by-step, practical guide to unleash the power of open processes and technology working together."

–Jim Whitehurst, President, IBM

"This book is packed with wisdom from Tim, Mike, Noel, and Donal and lovingly illustrated by Ilaria. Every principle and practice in this book is backed by wonderful stories of the people who were part of their learning journey. The authors are passionate about visualizing everything and every chapter is filled with powerful visual examples. There is something for every reader and you will find yourself coming back to the examples time and again."

–Jeremy Brown, Chief Technology Officer/Chief Product Officer at Traveldoo, an Expedia Company

"This book describes well what it means to work with Red Hat Open Innovation Labs, implementing industrial DevOps and achieving business agility by listening to the team. I have experienced this first hand. Using the approach explained in this book, we have achieved a level of collaboration and engagement in the team we had not experienced before, the results didn't take long and success is inevitable. What I have seen to be the main success factor is the change in mindset among team members and in management, which this approach helped us drive."

–Michael Denecke, Head of Test Technology at Volkswagen AG

"This book is crammed full to the brim with experience, fun, passion, and great practice. It contains all the ingredients needed to create a high performance DevOps culture...it's awesome!"

–John Faulkner-Willcocks, Head of Coaching and Delivery Culture, JUST

"DevOps has the opportunity to transform the way software teams work and the products they deliver. In order to deliver on this promise, your DevOps program must be rooted in people. This book helps you explore the mindsets, principles, and practices that will drive real outcomes."

–Douglas Ferguson, Voltage Control Founder, Author of Magical Meetings and Beyond the Prototype

"Fun and intense to read! Somehow, the authors have encapsulated the Red Hat culture and expression in this book."

–Jonas Frydal, Director at Volvo Cars

"This book is really valuable for me. I was able to map every paragraph I read to the journey we took during the residency with Red Hat Open Innovation Labs. It was such an intense but also rewarding time, learning so much about culture, openness, agile and how their combination can make it possible to deliver crucial business value in a short amount of time.

Speaking from my personal experience, we enabled each other, my team bringing the deep knowledge in the industry and Red Hat's team bringing good practices for cloud-native architectures. This made it possible to reinvent how vehicle electronics technology is tested while pushing Red Hat's OpenShift in an industrial DevOps direction.

I am looking forward to keeping a hard copy of the book at my desk for easy review."

—Marcus Greul, Program Manager at CARIAD, a Volkswagen Group company

"Innovation requires more than ideas and technology. It needs people being well led and the 'Open Leadership' concepts and instructions in DevOps Practice and Culture with OpenShift should be required reading for anyone trying to innovate, in any environment, with any team."

—Patrick Heffernan, Practice Manager and Principal Analyst, Technology Business Research Inc.

"Whoa! This has to be the best non-fiction DevOps book I've ever read. I cannot believe how well the team has captured the essence of what the Open Innovation Labs residency is all about. After reading, you will have a solid toolbox of different principles and concrete practices for building the DevOps culture, team, and people-first processes to transform how you use technology to act as a force multiplier inside your organization."

—Antti Jaakkonen, Lean Agile Coach, DNA Plc

"Fascinating! This book is a must-read for all tech entrepreneurs who want to build scalable and sustainable companies. Success is now handed to you."

—Jeep Kline, Venture Capitalist, Entrepreneur

"In a digital-first economy where technology is embedded in every business, innovation culture and DevOps are part and parcel of creating new organizational values and competitive advantages. A practical and easy to understand guide for both technology practitioners and business leaders is useful as companies accelerate their ***Digital Transformation*** **(DX)** *strategies to thrive in a changed world."*

—Sandra Ng, Group Vice President, ICT Practice

"DevOps Culture and Practice with OpenShift is a distillation of years of experience into a wonderful resource that can be used as a recipe book for teams as they form and develop, or as a reference guide for mature teams as they continue to evolve."

—David Worthington, Agile Transformation Coach, DBS Bank, Singapore

Get Your Own Copy

Buy the paperback from Packt Publishing for a quick and easy reference.

DevOps Culture and Practice with OpenShift

Copyright © 2021 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author(s), nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Authors: Tim Beattie, Mike Hepburn, Noel O'Connor, and Donal Spring

Illustrator: Ilaria Doria

Technical Reviewer: Ben Silverman

Managing Editors: Aditya Datar and Siddhant Jain

Acquisitions Editor: Ben Renow-Clarke

Production Editor: Deepak Chavan

Editorial Board: Vishal Bodwani, Ben Renow-Clarke, Edward Doxey, Alex Patterson, Arijit Sarkar, Jake Smith, and Lucy Wan

First Published: July 2021

Production Reference: 1100821

ISBN: 978-1-80020-236-8

Published by Packt Publishing Ltd.
Livery Place, 35 Livery Street,
Birmingham, B3 2PB, UK.

www.packt.com