Eötvös Loránd University

Faculty of Informatics

Dept. of Media

Educational Technology

Automated testing of applications with graphical user interface in cloud-native environments

Supervisor:
Horváth Győző
Associate Professor

Author:
Molnár Líviusz
Computer Science BSc

Budapest, 2023

Contents

Chapter 1

Introduction

As a DevOps engineer, I have seen the value and challenges of ensuring services that are reliable, scalable, and of high quality. Businesses are quickly integrating cloud-native technologies, using microservice designs to improve reliability and scalability. Consequently, modern systems are far more complicated and have many more moving elements.

Pre-deployment unit, integration, system, acceptance testing, and live monitoring with logs, metrics, traces, and alarms ensure high quality. However, most technologies can only record particular mistakes, making them unable to quantify the entire user experience.

Selenium offers the toolkit needed to implement system and acceptability level tests by simulating user behaviour. It can only be used during the testing phase prior to deployment, and the existing live production monitoring cannot reliably identify faults amongst system components.

By developing a go-based Kubernetes operator that automates selenium test runs and aggregates the results in Prometheus format, my thesis project aims to bring selenium's capabilities to the live production cloud-native space while also offering a simple-to-use instrument to integrate selenium testing into industry-standard monitoring/alerting chains.

Chapter 2

User documentation

The ability to improve the scalability, agility, and dependability of applications in cloud environments has led to a rise in the popularity of cloud-native development in recent years. The foundation of this strategy is containerization, which offers a standardized and portable way to package and distribute software.

The industry standard for scaling up containerized application management is Kubernetes, a well-known container orchestration technology. In addition to tools for automating these procedures, it offers a complete set of primitives for deploying, scaling, and managing containerized applications.

For automated web browser testing of online applications, particularly in continuous integration and continuous delivery (CI/CD) pipelines, Selenium, an open-source testing framework, is crucial. In a cloud-native setting, Selenium testing integration with Kubernetes is essential.

This project implements a Go-based Kubernetes operator, a custom controller that extends the Kubernetes API and automates the deployment and management of Selenium tests in a Kubernetes cluster while making it easy to integrate into industry-standard monitoring and alerting chains.

This documentation will cover all operator usage aspects, including installation, configuration, and customization. We will also provide examples of how to use the operator to automate the execution of Selenium tests and how to integrate it with other Kubernetes-native tools and services.

2.1 Prerequisites of employing the operator

The operator is designed to be utilized in Kubernetes clusters of business scale, whether they are hosted on public cloud servers or on-premise servers. Although the operator was designed with performance in mind, neither Kubernetes, Selenium hubs nor Prometheus monitoring stacks are intended for local development; however, they are all necessary to show the project's feasibility. Since this tutorial will show it on a local Kubernetes cluster, be aware that hardware-related issues could occur.

Requirements are the followings:

- Kubernetes cluster (minikube in the manual)
- Shell environment (Linux/Mac OS or Windows with WSL)
- kubectl CLI tool
- make CLI tool
- selenium test exported to .side format
- rented or self-managed Selenium Hub service (moon in the manual)
- operator-managed Prometheus in the Kubernetes cluster (kube-prometheus in the manual)
- Helm CLI tool for deploying all the above

This manual will demonstrate setting up the following Kubernetes environment before deploying the first automated test:

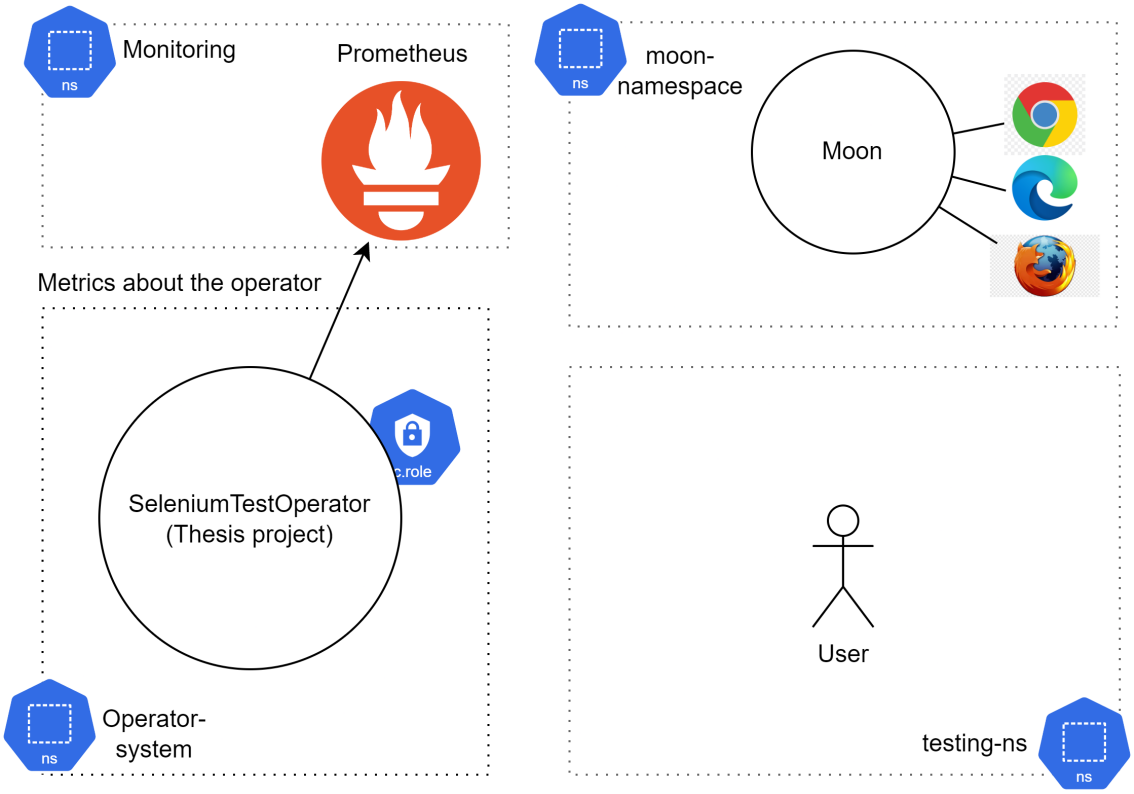

Figure 2.1: Architecture without running tests

2.1.1 Installing a Minikube cluster

Before setting up the tools required for running Selenium tests with Kubernetes, a few prerequisites need to be installed. These prerequisites include a Kubernetes cluster, a virtualization tool for Minikube, a shell environment, the kubectl CLI tool, the make CLI tool, and a Selenium test exported to .side format. In this guide, we'll go through each of these prerequisites in detail and show you how to install them on your machine. Once these prerequisites are in place, we can move on to setting up the SeleniumTest Operator and creating custom resources for running Selenium tests.

Before everything else, Windows users has to install WSL by opening the Microsoft Store on Windows and search for "Ubuntu" or any other Linux distribution of choice and launch it.

From now on, shell environment will be used universally between operating systems, meaning shell on either Linux, Mac OS or Windows' WSL.

- First of all, intall kubectl:

- In the shell terminal, run the following commands:

```
curl -LO "https://storage.googleapis.com/kubernetes-release/release/$(curl -s https://storage.googleapis.com/kubernetes-release/release/stable.txt)/bin/linux/amd64/kubectl"
chmod +x ./kubectl
sudo mv ./kubectl /usr/local/bin/kubectl
```

Code 2.1: Installing kubectl

- Verify that kubectl is installed:

```
kubectl version --short
```

Code 2.2: Verifying kubectl

- Secondly, to install make, run the following command in the shell terminal:

```
sudo apt-get install build-essential
```

Code 2.3: Installing make

- Thirdly, to install helm, run In the shell terminal:

```
curl -fsSL -o get_helm.sh https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3
chmod 700 get_helm.sh
./get_helm.sh
```

Code 2.4: Installing helm

- Fourthly, to install and configure Minikube:

  - Windows users:

    * Open your web browser and go to the Minikube releases page on GitHub (https://github.com/kubernetes/minikube/releases).
    * Find the latest release for Windows and download the executable file (e.g., minikube-windows-amd64.exe).
    * Move the downloaded executable to a folder that is included in your PATH environment variable. For example, you can move it to the "C:

Windows

System32" folder.

* Install Docker desktop for Minikube from its offical site: https://www.docker.com/products/docker-desktop/

– Linux users:

```
curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
sudo install minikube-linux-amd64 /usr/local/bin/minikube
```

Code 2.5: Installing minikube cli for Linux

– Mac users:

* Install Homebrew by running the following command in Terminal:

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

Code 2.6: Installing homebrew cli for Mac

* Once Homebrew is installed, you can install minikube by running the following command in Terminal:

```
brew install minikube
```

Code 2.7: Installing minikube cli for Mac

– After the installation is complete, validate it with the following command:

```
minikube version
```

Code 2.8: Verifying minikube cli tool

- Lastly, we need to start Minikube:

– Minikube startup under Windows:

```
minikube start --cpus=4 --memory=8G --disk-size=30G --driver=docker
```

Code 2.9: Starting minikube for Windows

  - Minikube startup under MacOS:

```
minikube start --cpus=4 --memory=8G --disk-size=20G
    --driver=hyperkit
```

Code 2.10: Starting minikube for MacOS

  - Minikube startup under Linux

```
minikube start --cpus=4 --memory=8G --disk-size=20G
    --driver kvm2
```

Code 2.11: Starting minikube for Linux

- Even though Moon advises against using Docker as a driver, it is necessary from a networking perspective for Windows users in order to be able to access the cluster from WSL. Providing Minikube with more resources is strongly recommended if possible, but these should be the absolute minimum.
- Verify that Minikube is running:

```
kubectl cluster-info
```

Code 2.12: Verifying minikube cluster

2.1.2 Solving kubectl connection problem on Windows

A configuration file for the kubectl command-line tool for Kubernetes can be found at

```
~/.kube/config
```

In addition to the credentials required for cluster authentication, it indicates the Kubernetes cluster with which kubectl should communicate.

Depending on the setup, when installing Minikube, it creates a Kubernetes cluster in a virtual machine or a container. Minikube changes the computer's

```
~/.kube/config
```

file to set up kubectl to use this new cluster automatically. As a result, kubectl can communicate with the Minikube cluster.

Windows users encounter a problem since Minikube sets the configuration file at the Windows operating system level, whereas kubectl refers to the configuration file contained in the WSL distribution.

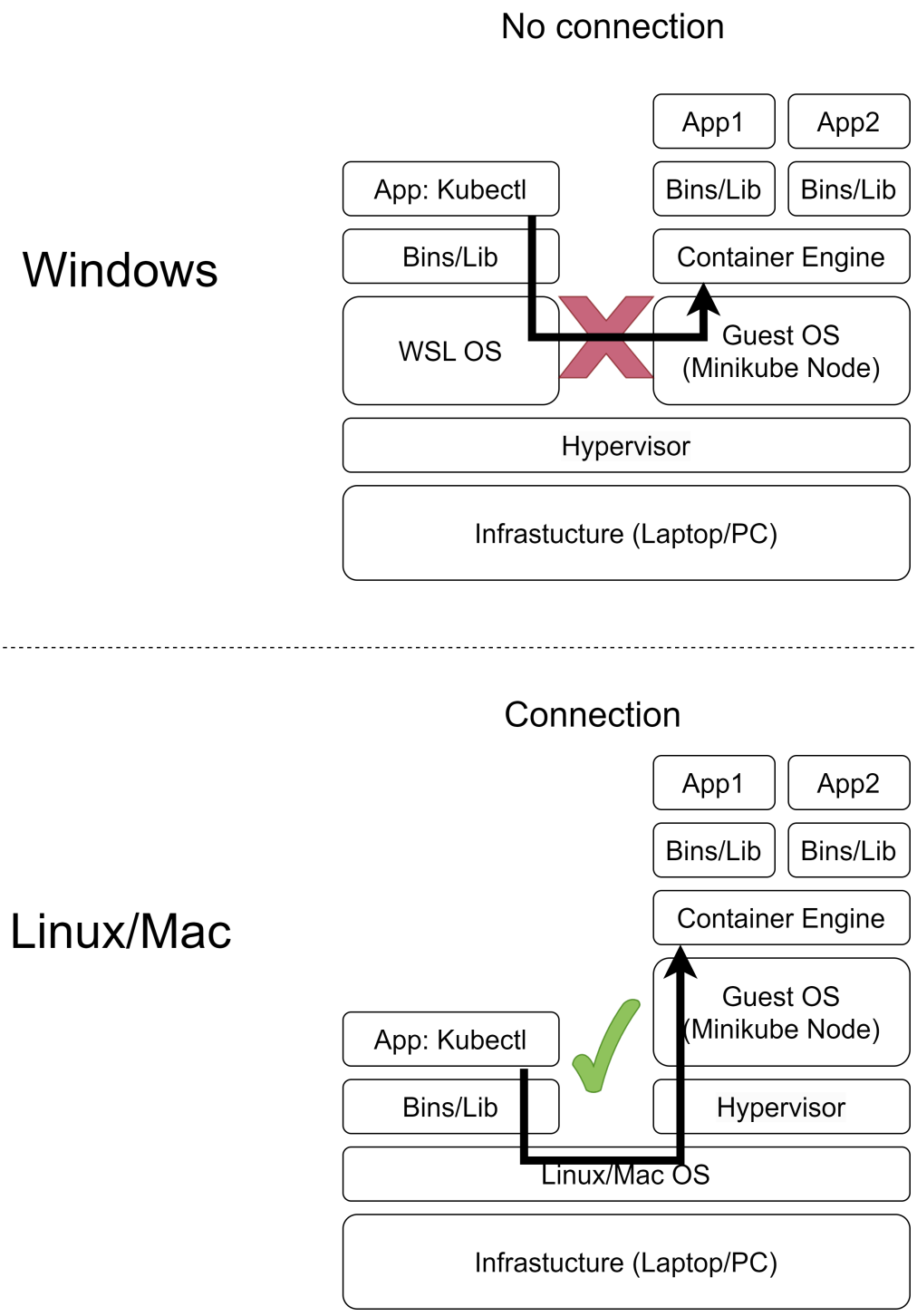

Figure 2.2: Virtualization difference between WSL and Unix systems

We must manually generate the kubeconfig file based on the one Minikube created to fix the problem. Although the two files will be very similar, there will be a few minor adjustments because one was created for Windows and the other for Linux.

This is how the new kubeconfig file should appear:

```
apiVersion: v1
clusters:
- cluster:
    certificate-authority: /mnt/c/Users/Your_User/.minikube/ca.crt
    extensions:
    - extension:
        last-update: Sat, 18 Mar 2023 17:26:55 CET
        provider: minikube.sigs.k8s.io
        version: v1.27.0
        name: cluster_info
    server: https://127.0.0.1:8443
    name: minikube
contexts:
- context:
    cluster: minikube
    extensions:
    - extension:
        last-update: Tue, 02 May 2023 22:28:36 CEST
        provider: minikube.sigs.k8s.io
        version: v1.27.0
        name: context_info
    namespace: default
    user: minikube
    name: minikube
current-context: minikube
kind: Config
preferences: {}
users:
- name: minikube
    user:
    client-certificate: /mnt/c/Users/Your_User/.minikube/profiles/
        minikube/client.crt
    client-key: /mnt/c/Users/Your_User/.minikube/profiles/minikube/
        client.key
```

Code 2.13: Example kubeconfig file

After creating the file, run the following command in the WSL terminal:

```
export KUBECONFIG=path/to/your/kubeconfig
```

Code 2.14: Setting Kubeconfig

Lastly, verify that Minikube is reachable:

```
kubectl cluster-info
```

Code 2.15: Validate Minikube

2.1.3 Deploying the required applications to the cluster

This section will set up the kube-prometheus, Moon, and seleniumTest operator on our Kubernetes cluster. These tools are essential for testing and monitoring our cluster to ensure it runs smoothly and dependably. For our cluster, kube-prometheus offers a reliable monitoring solution that enables us to keep an eye on critical parameters like resource utilization and application performance. While the SeleniumTest Operator automates the deployment and upkeep of those tests in our Kubernetes cluster, Moon is a Selenium Grid implementation that we will use to run our Selenium tests.

We will only deploy the Prometheus operator portion of the kube-prometheus stack because of the limited hardware resources available locally, as the other components are useless for this demonstration. Enter the operator repository to get started, then execute the following commands:

```
kubectl apply --server-side -f prometheus/setup
kubectl wait --for condition=Established --all
    CustomResourceDefinition --namespace=monitoring
kubectl apply -f prometheus/
```

Code 2.16: Deploying Prometheus operator

Whenever there are issues, visit kube-prometheus's official github page for more details on how to deploy it: https://github.com/prometheus-operator/kube-prometheus

The following commands will deploy Moon, the Selenium Grid implementation, so that the tests may be run:

```
helm repo add aerokube https://charts.aerokube.com/
helm repo update
kubectl create namespace moon
helm upgrade --install -f moon_values.yaml -n moon moon aerokube/
    moon2
```

Code 2.17: Installing moon

Ingress controller is disabled in the moon$_v alues.yaml file because we won't attempt to contact Moon fr

The seleniumTest operator can then be deployed; perform the following steps from the operator's repository:

```
make deploy IMG=quay.io/molnar_liviusz/selenium-test-operator:v0
    .0.24
```

Code 2.18: Deploying of operator

Finally, we need to create and change the context to the namespace for our project, where we will launch all of the Selenium tests:

```
kubectl create namespace testing-ns
kubectl config set-context --current --namespace=testing-ns
```

Code 2.19: Creating testing namespace

Downloading Lens is advised for simpler cluster management (https://k8slens.dev/). Lens offers a straightforward and understandable graphical user interface for controlling Kubernetes clusters. Users can navigate and change resources, view their cluster's current state, and keep an eye on performance indicators in real time. Additionally, Lens supports numerous clusters and offers a single location for managing them all. Lens is a fantastic alternative for developers and system administrators who need to work with Kubernetes on a regular basis due to its user-friendly interface and extensive feature set.

The outcome of all this is that we now have our cluster configured, achieving the cluster state shown in the previous architecture diagram, with the seleniumTest operator, Prometheus, to query our operator's metrics (including the test results), and Moon deployment as a Selenium Grid implementation. The test recordings are the final component needed to deploy tests.

2.1.4 Recorded selenium tests with Selenium IDE

Selenium IDE is a browser extension that may be used to record user interactions with a web application. The user can export the recorded interactions in the ".side" (Selenium Integrated Development Environment) file format once the recording is finished. This file may then be used with Selenium WebDriver to reliably and repeatedly automate the same interactions. The ".side" file provides details on the actions taken, the web page elements that were the targets, and any additional data

like input values or anticipated outcomes. Developers and testers can save time and effort while creating and maintaining automated tests by utilizing Selenium IDE to record and export interactions into a ".side" file.

To create a selenium test, open the extension > Record a new test in a new project > add a project name > add the URL where the browser should open at the beginning of the test > start recording > finish recording:

Figure 2.3: SeleniumIDE 1

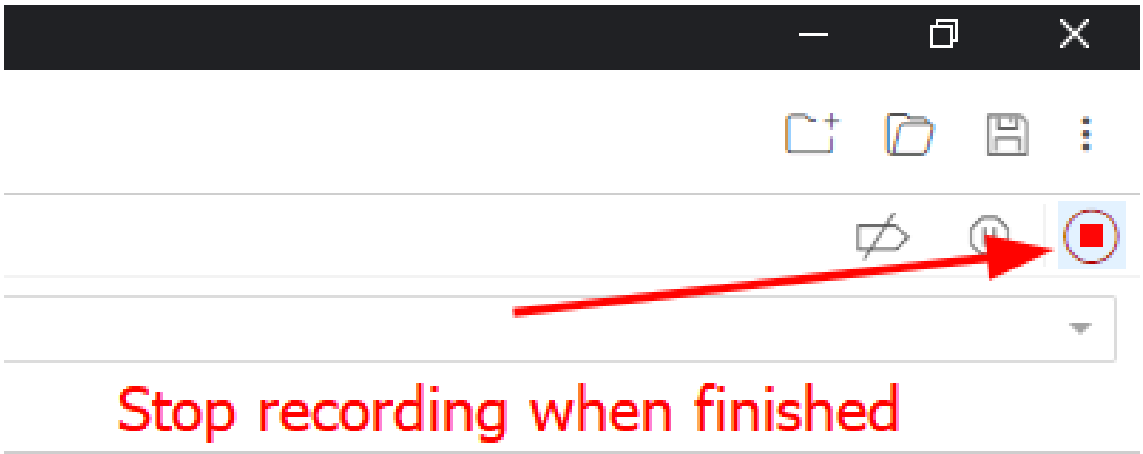

Figure 2.4: SeleniumIDE 2

Replacing the actions' default target variables with a specific XPath target is advised to increase test reliability. Recommend doing this while utilizing the "XPath

Helper" browser plugin and the browsers' inspect mode:

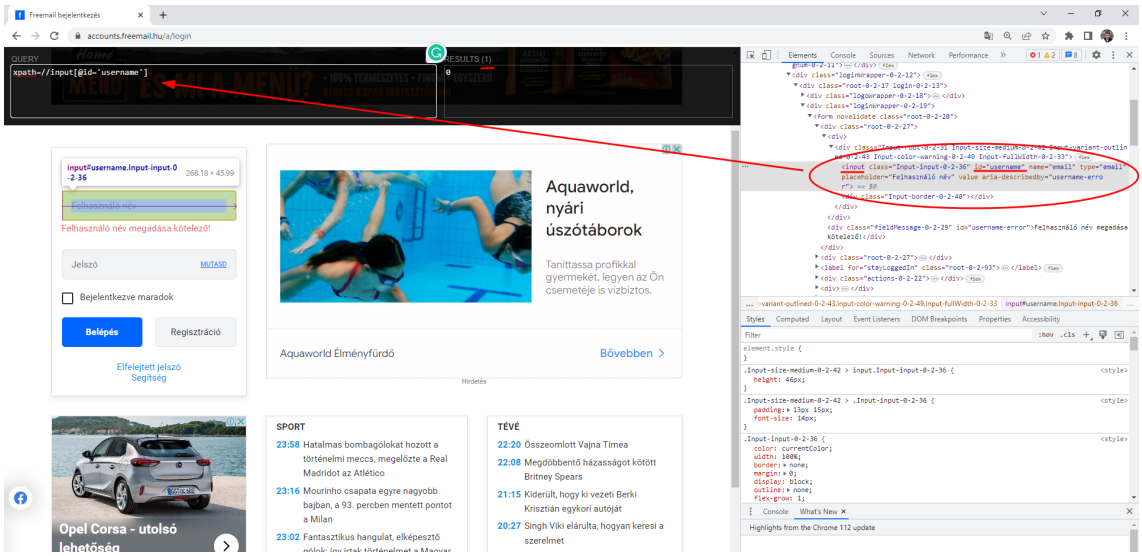

Figure 2.5: XPath Helper

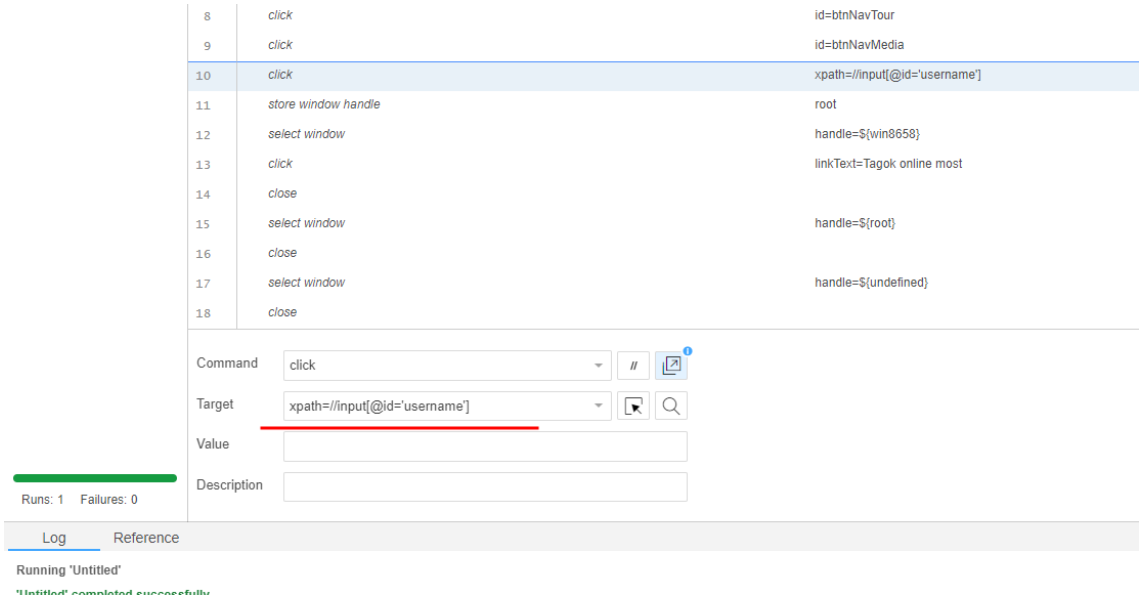

Figure 2.6: SeleniumIDE 3

Several tests can be conducted within a single test project, and sophisticated Selenium Grid implementations, such as Moon, can perform these tests in a paralyzed fashion to increase efficiency and shorten the testing time. Once finished, save the project into the default .side extension.

2.2 Deploying Automated Selenium Tests

Automating the running of Selenium tests on a Kubernetes cluster requires a configuration file for the test, a ConfigMap that stores the .side file. A custom resource called SeleniumTest needs to be created to define the desired state of the test, such as the image to be used, the Selenium Grid to connect to, and the schedule for running the test. With these two components in place, Kubernetes can automatically run the Selenium test according to the specified schedule, greatly simplifying web application testing.

The configmap looks resembles the following:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: testcode
  namespace: testing-ns
data:
  # file-like keys
  exampletest.side: |
  {
    "id": "02faada0-52b8-49fe-95cc-d403a5ef9dcd",
    "version": "2.0",
    "name": "Operator",
    "url": "https://accounts.freemail.hu",
    "tests": [{
    "id": "cf677198-9658-455c-b524-c2fa55ad59f7",
    "name": "login",
    "commands": [{
      "id": "31de9648-b82a-4322-90da-65ee6a89a9f5",
      .
      .
      .
```

Code 2.20: Example configmap with .side test

The following is the syntax for the seleniumTest custom resource:

```
apiVersion: selenium.mliviusz.com/v1
kind: SeleniumTest
metadata:
  labels:
    app.kubernetes.io/name: seleniumtest
```

```
      app.kubernetes.io/instance: seleniumtest-sample
      app.kubernetes.io/part-of: operator
      app.kubernetes.io/managed-by: kustomize
      app.kubernetes.io/created-by: operator
    name: seleniumtest-sample
    namespace: testing-ns
  spec:
    schedule: "*/2 * * * *"
    repository: quay.io
    image: molnar_liviusz/selenium-test-runner
    tag: v0.0.10
    configMapName: testcode
    retries: "3"
    seleniumGrid: "http://moon.moon.svc:4444/wd/hub"
```

Code 2.21: SeleniumTest CR yaml format

The SeleniumTest object's desired state is specified in the spec section of the custom resource with the following variables:

- repository: specifies the repository for the Docker image.
- image: The Docker image that will be used to execute the test is specified by this option. This image is also part of the thesis project
- tag: This parameter specifies the tag for the Docker image.
- retries: The number of times the test should be rerun if it fails is specified by this option. The test counts as a pass even if one is successful.
- schedule: This parameter specifies the schedule for the CronJob that runs the test.
- seleniumGrid: This parameter specifies the URL of the Selenium Grid for running the test.

To deploy the test, use the following commands:

```
kubectl apply -f testfiles/testcode-configmap.yaml
kubectl apply -f testfiles/sample-seleniumtest.yaml
```

Code 2.22: Deploying example tests

Following the previous steps, the test results can be viewed on Prometheus, where 0 means succes, 1 means the test failed:

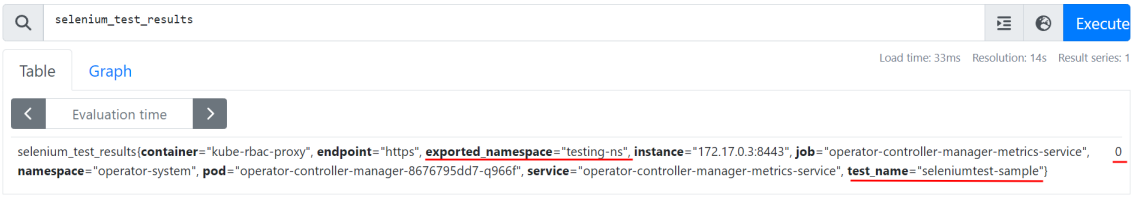

Figure 2.7: Seleniumtest metric within Prometheus

Chapter 3

Developer documentation

This essay automates Selenium tests in a cloud-native environment using a Go-based Kubernetes operator. This section contains full usage instructions for this Kubernetes operator and in-depth explanations of its features, design philosophy, and supporting technologies.

The Kubernetes operator is a tool that automates the deployment, maintenance, and scaling of Selenium tests in a Kubernetes cluster, which will be covered in this documentation. It makes it simple for DevOps teams and developers to integrate Selenium testing into production workflows, guaranteeing that their cloud-native applications are extensively tested and optimized for performance, scalability, and dependability.

This documentation will provide all the information to know to take advantage of the capabilities of this Go-based Kubernetes operator for Selenium testing, whether you are an experienced Kubernetes developer or are just getting started with cloud-native application development. Therefore, let us get in and get to exploring!

3.1 Understanding the underlying technologies

Understanding the underlying technologies that enable the Go-based Kubernetes operator for Selenium testing is crucial before we go into its specifics. The necessity of introducing these technologies and their importance in the context of developing cloud-native applications will be covered in this section.

Due to its capacity to increase the scalability, agility, and dependability of applications running in a cloud environment, cloud-native application development has

grown in popularity in recent years. The use of containerization, which offers a standardized and portable mechanism to bundle and distribute programs, is at the core of this strategy.

Popular container orchestration platform Kubernetes has become the de facto standard for scalably managing containerized applications. In addition to tools for automating these procedures, it offers a complete set of primitives for deploying, scaling, and managing containerized applications.

On the other hand, the open-source testing framework Selenium enables programmers to automate web browser testing for web applications. In the context of continuous integration and continuous delivery (CI/CD) pipelines, in particular, it has evolved into a crucial tool for maintaining the quality and performance of online applications.

It is vital to integrate Selenium testing with Kubernetes to use its advantages in a cloud-native environment. The Go-based Kubernetes operator is helpful in this situation. To automate the deployment and maintenance of Selenium tests in a Kubernetes cluster, a custom controller that extends the Kubernetes API is used.

Understanding the underlying technologies is crucial for successfully operating the Go-based Kubernetes operator for Selenium testing. By doing this, DevOps teams and developers may better understand the utility of this operator and utilize it to improve their processes for creating cloud-native applications.

3.1.1 Differences between Virtual Machines and Containers

Virtual machines and containers are two standard technologies in cloud computing and application deployment. Virtual machines and containers offer isolation and virtualization, but they differ in several important ways. The distinctions between virtual machines and containers are covered in this section.

Software-based simulations of physical devices are known as virtual machines. They execute a whole operating system on top of a hypervisor, emulating the entire hardware stack, including the processor, memory, and storage. Each virtual machine has its own virtualized hardware, operating system, and isolated environment in which it operates. As a result, it is possible to run many operating systems on a single physical machine and achieve total isolation.

On the other hand, containers are a minimal type of virtualization that works at the operating system level. Multiple applications can run on the same operating system instance thanks to containers, which use the host operating system kernel and share resources with other containers. Virtual machines can be replaced by more lightweight and practical containers, which speed up the deployment and scaling of applications.

One of the key distinctions is how virtual machines and containers use their resources. Resource wastage can occur because virtual machines need many resources to replicate a whole hardware stack. Contrarily, containers use resources better because they share the host operating system kernel.

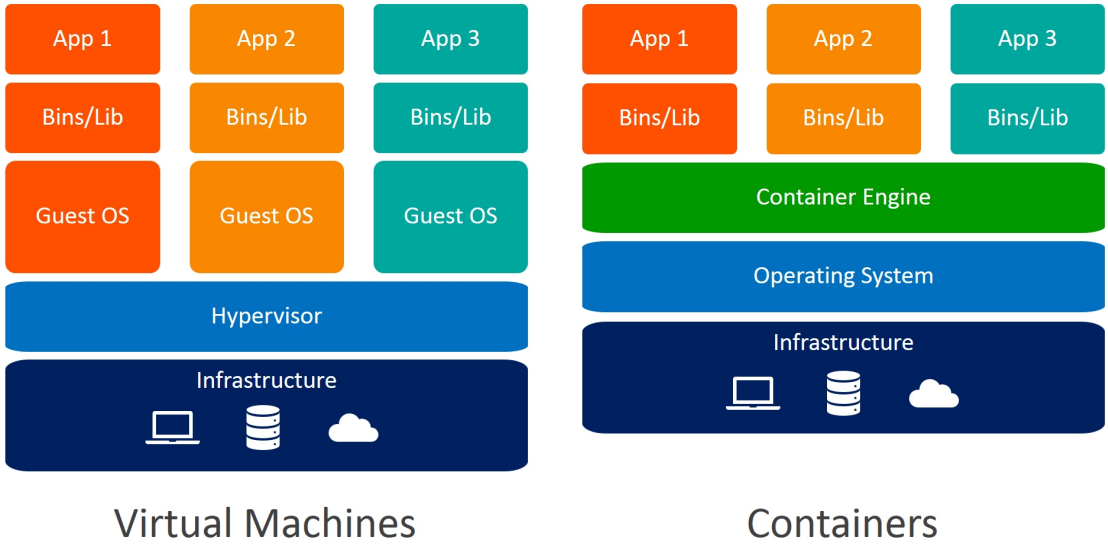

Figure 3.1: Containers vs Virtual Machines

The starting times of virtual machines and containers are another distinction. Virtual machines typically take longer to start up because a complete operating system must be booted. In contrast, containers start up relatively instantaneously because just the containerized program needs to be started.

In conclusion, although virtual machines and containers offer virtualization and isolation, they differ in resource usage, startup time, and security, with containers providing a more portable and practical option for running applications.

3.1.2 The Advantages of Container Orchestration with Kubernetes

Automating the deployment, scaling, and maintenance of containerized applications is known as container orchestration. In the age of cloud computing, container orchestration has become an essential tool for managing massive applications. The most well-known and commonly used container orchestration platform is Kubernetes.

The ability to automatically deploy and scale containerized apps is one of the critical benefits of Kubernetes-based container orchestration. Developers can declare the intended state of their application using Kubernetes' declarative configuration technique, and Kubernetes will take care of the rest. By doing this, the application can scale up or down as necessary and will continuously operate as intended.

Additionally, a variety of built-in features and extensions provided by Kubernetes enable sophisticated container orchestration capabilities. These include, among other things, automatic failover, load balancing, advanced scheduling, and service discovery. Kubernetes also offers a powerful command-line interface and API for communicating with and administering containerized applications.

Kubernetes' capacity to function on any cloud infrastructure or on-premises data center is a substantial additional benefit. No matter where containerized applications are running, Kubernetes abstracts away the underlying infrastructure and offers a standard application deployment and management method. Because of this, fully cloud-native apps that can be deployed anywhere may be created.

Additionally, Kubernetes has a sizable and vibrant open-source community, which means it is continuously developing and getting better. The community updates Kubernetes with additional functions, bug fixes, and security patches, transforming it into a dependable and safe environment for running mission-critical applications.

In conclusion, Kubernetes container orchestration is necessary for managing modern cloud-native applications. It is the best platform for managing containerized applications due to its powerful orchestration features, automation of deployment and scaling, and flexibility to run on any infrastructure. Thanks to its vibrant and expanding community, it will continue to be a safe and dependable platform for managing apps for many years.

3.1.3 Fundamental Kubernetes Objects

The open-source container orchestration platform Kubernetes automates containerized applications' deployment, scaling, and maintenance. Kubernetes enables developers to construct and manage containerized applications at scale regardless of the underlying infrastructure.

A pod, a single instance of a container running on a node, is the smallest deployment unit in Kubernetes. One or more containers may be included in pods, which share the same network namespace and storage. By guaranteeing that a certain number of replicas, or exact duplicates of the pod, are active at all times, replicasets control the scaling of pods.

A set of replicasets and pods may be managed in the desired state declaratively using deployments. By establishing a deployment and describing the number of copies and the containers' desired state, developers can ensure the application is functioning as planned. Rolling updates, rollbacks, and version control are additional features of deployments.

By providing a consistent IP address and DNS name for accessing a group of replicas, services isolate network access to pods and replicasets. Services can expose pods openly to the internet or internally to other pods in the same namespace.

Objects in a Kubernetes cluster are grouped using namespaces. They give a mechanism to group and protect resources, so one can use them to separate teams or applications into logical partitions inside a cluster.

Configmaps hold configuration information that running containers can access. They offer a practical method for handling application configuration information, including environment variables or configuration files.

Jobs are Kubernetes objects that execute a specified task, such as a batch job or a script, until it is finished. CronJobs, on the other hand, are objects that execute a specific operation at a predetermined time, such as executing a backup script every day at midnight.

These Kubernetes components offer a robust and adaptable infrastructure for managing containerized applications. Kubernetes allows developers to concentrate on developing code and providing business value rather than worrying about the underlying infrastructure by utilizing these abstractions

and automating typical tasks.

3.1.4 Kubernetes custom resources and operators

Developers can control many facets of their application deployments using the wide range of built-in resources that Kubernetes offers. However, some applications need specialized resources that Kubernetes does not ship with. Kubernetes provides custom resources, enabling programmers to create API objects representing unique resource types.

Because they can be used to represent any resource that is not already included in Kubernetes, custom resources are crucial. These unique resources can be used to specify certain actions, like configuring a firewall just for an application or creating a unique scaling policy. Custom resources can be generated, maintained, and watched over in the same ways as built-in resources.

Making a unique resource is simply the first action in any case. Developers frequently need bespoke controllers in order to manage customized resources efficiently. A Kubernetes operator known as a controller tracks changes to a custom resource and initiates the necessary responses. The operator carries out a variety of duties, like changing a configuration, starting a deployment, or keeping track of a resource.

An efficient technique to automate complicated activities involving bespoke resources is with Kubernetes operators. Developers can design operators to give particular behavior for a unique resource. As a result, operators automate resource management rather than managing resources by hand, lowering the possibility of human error and speeding up the process.

The ability for developers to customize Kubernetes to meet their unique needs is one of the advantages of its bespoke resources and operators. Instead of constraining their workflows to fit into Kubernetes' existing architecture, developers can design new resources and operators to automate their unique workflows.

To sum up, Kubernetes' custom resources and operators are robust tools that give developers the ability to increase the functionality of Kubernetes to match their particular needs. Developers can define their API objects for custom types of resources using custom resources, and operators can automate complex activities involving these resources. Developers can create a more robust and automated

Kubernetes environment that suits their unique requirements by employing custom resources and operators.

3.1.5 Understanding Kubernetes Role-Based Access Control (RBAC)

An essential security component of Kubernetes is role-based access control (RBAC), which enables users to specify explicit permissions and access policies for managing Kubernetes resources. Kubernetes administrators can control who has access to and can alter resources at a precise level with RBAC.

By establishing roles and bindings, RBAC enables Kubernetes cluster managers to manage access to the cluster. A binding ties a role to a user, group, or service account, whereas a role defines a set of rules that specify the actions a user or group can do on Kubernetes resources. Kubernetes administrators can build bespoke access policies that perfectly meet their security needs by combining roles and bindings.

Cluster roles, cluster role bindings, roles, and roles bindings are only a few of the roles and bindings the RBAC feature in Kubernetes provides. While roles and role bindings define permissions for resources inside a namespace, cluster roles, and cluster role bindings define permissions for resources across the entire cluster.

The definition of cluster-wide permissions that can be granted to individuals or groups uses cluster roles and cluster role bindings. Cluster roles are created by providing a set of rules that specify the operations that can be carried out on a resource. A user, group, or service account is bound to a cluster role by a cluster role binding. The resources indicated in the cluster role can now be accessed and modified by the user or group.

Namespace-level permissions are defined using roles and role bindings and can be given to individuals or groups. Roles are built by specifying a set of rules that limit what can be done to resources within a given namespace. By associating a role with a user, group, or service account, those accounts can access and modify resources in the namespace designated by the role.

Granular control over who can access and alter resources within a Kubernetes cluster is possible thanks to Kubernetes RBAC. RBAC allows cluster administrators to handle rights at a highly granular level, which is crucial for big businesses with detailed security requirements.

3.1.6 Microservice Architecture and Kubernetes: A Perfect Match for Scalability and Agility

Creating an application as a collection of loosely connected, independently deployable services is known as microservice architecture. Typically, each microservice only handles one task and interacts with other microservices using clear APIs.

The popularity of the microservice architecture can be attributed to a number of factors. The ability for teams to build, test, and deploy separate services independently is one of its key benefits. These can lead to shorter development cycles and more agility. Additionally, because microservices can be scaled horizontally, more service instances can be added to meet growing demand, improving the performance and scalability of the entire application.

Because it offers many features that make it simple for teams to deploy, manage, and scale microservices, Kubernetes, as a container orchestration platform, is a good fit for the microservice architecture. By enabling teams to deploy microservices as containers, Kubernetes offers a dependable and adaptable environment that can run on any infrastructure. Additionally, Kubernetes has functions like load balancing, service discovery, and automatic scaling essential for managing large-scale microservices applications.

In conclusion, microservice architecture is favored because it improves software development's flexibility, scalability, and agility. Microservices may be quickly launched, managed, and scaled to meet the demands of modern applications when used in conjunction with Kubernetes. Microservices and Kubernetes working together can speed up application development cycles, enhance application performance, and boost resilience and fault tolerance.

3.1.7 Challenges of Testing Microservices and How Selenium Helps

Although the flexibility and scalability of the microservice architecture are two of its many advantages, it also presents some unique testing challenges. Due to their distributed nature, the complexity of the system, and the requirement to ensure that all services function together without interruption, testing microservices thoroughly

can be challenging. Fortunately, there are tools like Selenium that can assist in overcoming some of these obstacles.

The fact that processes are spread across several services makes it difficult to test individual services, which is one of the critical challenges with testing microservices. Additionally, it can be challenging to guarantee that each service functions correctly because adjustments to one service may impact how another behaves. Furthermore, microservices frequently use external APIs, which makes testing more challenging.

The system's intricacy poses another difficulty. It can be challenging to keep track of what is happening and where problems occur when numerous small services are communicating. This complexity can cause delays in problem resolution by making it challenging to pinpoint the source of a problem.

An open-source automated testing tool called Selenium can assist in overcoming some of these difficulties. In order to make testing web applications that use microservices simpler, it offers a set of tools for automating web browsers. Testers can mimic user interactions with a web application using Selenium to ensure all services operate as intended.

Selenium can test an application more completely and faster than manual testing since it can automate testing. Additionally, it is capable of running tests concurrently, enabling quicker feedback and shorter testing cycles. Selenium also offers an interface with other testing frameworks and tools, such as JUnit and TestNG, which makes it simpler to incorporate into already-in-place testing procedures.

In conclusion, thorough testing might be difficult because of the dispersed nature and complexity of microservices. However, by automating testing and offering a suite of tools for testing web applications that rely on microservices, tools like Selenium can help to overcome some of these difficulties. Using Selenium, testers can check that microservices interact adequately and quickly spot problems, resulting in quicker feedback and shorter testing cycles.

3.1.8 An Overview of Selenium: Application Testing Tool

Selenium is a web application testing tool. It automates testing online applications by imitating user behaviors, including button clicks, text entries, and page navigation. Developers can use Selenium to create scripts that automate time-consuming testing processes, saving time and effort compared to manual testing.

Java, Python, C#, Ruby, and JavaScript are just a few programming languages supported by the open-source Selenium tool. Selenium WebDriver, Selenium Grid, and Selenium IDE are some of its various parts.

A programming interface called Selenium WebDriver enables programmers to automate interactions with web browsers. It offers a selection of techniques for manipulating online components like buttons, text boxes, and drop-down menus, moving between pages, and utilizing browser capabilities like tabs and windows.

Selenium Grid is a tool for simultaneously executing tests on several operating systems and browsers. It permits parallel testing, which can drastically cut down on the amount of time needed for testing.

The browser extension Selenium IDE offers a user-friendly interface for building and running Selenium tests. It enables developers to capture user interactions and produce test scripts without creating any code.

Selenium IDE stores test cases in files with .side extensions. They are JSON files that contain information about the test case, such as the name, description, and steps. The steps in a .side file describe the actions that Selenium should perform during the test, such as clicking a button or entering text into a text box.

As a result of imitating user activities, Selenium is a popular testing tool for web applications that automates the testing process. Selenium WebDriver, Selenium Grid, and Selenium IDE are some of its various parts. In order to store test cases, Selenium IDE uses .side files. It also offers a user-friendly interface for developing and running Selenium tests.

3.1.9 Prometheus, Alerting and Service Monitors

Popular open-source monitoring software Prometheus gathers, stores, and queries metrics from various sources. It offers a versatile query language and a potent data model that allow users to learn essential things about the functionality and state of their systems. However, more than simply gathering and storing metrics is required; (human) operators must also be informed when specific situations call for their attention. Prometheus alerting and service monitors are helpful in this situation.

Users can create alerting rules in Prometheus that send notifications when specific metrics exceed predetermined thresholds. Users can select from a wide variety of notification channels, including email, Slack, and PagerDuty, to name a few, and

these rules can be highly configurable. Additionally, users can use Prometheus' robust query language to develop more intricate alerting scenarios.

On the other hand, service monitors are Kubernetes resources that specify how Prometheus should monitor a particular service. They enable administrators to automatically find services operating in a Kubernetes cluster and set up Prometheus to monitor those services. Service monitors specify which metrics should be collected and how they should be collected. They can also include labels to help organize the data that has been gathered.

Alerting and service monitors work together to give operators the ability to create effective monitoring and alerting systems that can identify problems before they become critical and warn the necessary parties for a quick resolution.

3.2 The Operator: from Architecture to fine details

This section discusses the architecture of the thesis project operator, providing all the information necessary to fully comprehend how its many components relate to one another and how their interdependencies operate.

In a nutshell, the operator accepts the SeleniunTest kind of custom resource, specifying the name, scheduling, browser runner service, and config map holding the selenium .side test file. As a result, the operator constructs a cronjob with the necessary permissions, which arranges the tests to run on the browser runner service when executed at the specified times. The test results are subsequently written into a SeleniumTestResult type of CR. The operator responds with its reconciliation logic, which exposes the test results in Prometheus format in a central location for Prometheus-like monitoring solutions to integrate into their alerting chain. Each component will be more thoroughly explained in this section.

In addition, viable other integration methods will be highlighted, demonstrating how adaptably a company can integrate it into its environment without resorting to quirky workarounds or modifying the operator.

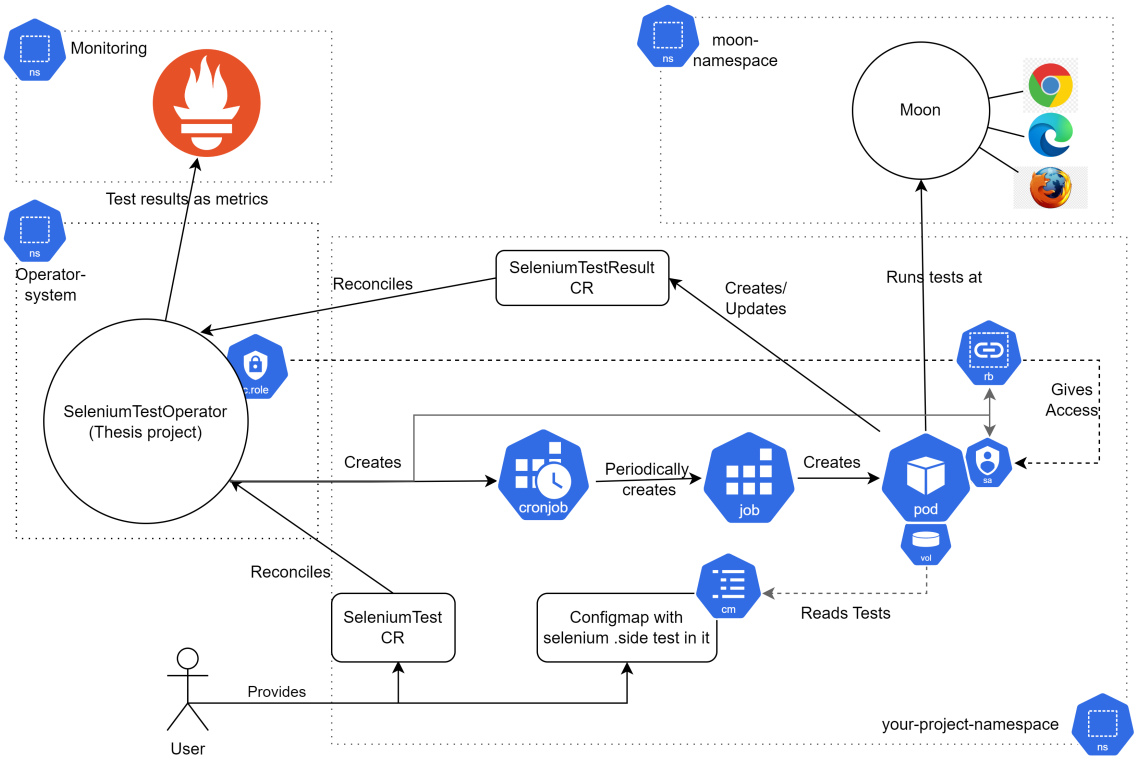

Figure 3.2: Whole Architecture

3.2.1 Comparing different Kubernetes operators

It is necessary to analyze the available possibilities and select the most appropriate one because Go-based operators are not the only way to develop a tool of this nature.

Helm-based operators oversee the deployment and lifetime of Kubernetes applications using Helm charts, as the name implies. They offer a high level of abstraction for managing complex applications and are comparatively simple to use. They may, however, be less adaptable than other kinds of operators.

Ansible-based operators use Ansible playbooks to control Kubernetes application deployment and lifecycle management. They offer great flexibility and are simple to include in current Ansible procedures. They can, however, be more challenging to set up and maintain.

The Kubernetes API is used by Go-based operators, such as the SeleniumTest operator in this guide, to control the deployment and lifecycle of apps. They offer outstanding flexibility and are simple to adapt to certain use situations. However, they can be more challenging to create and need a better comprehension of Kubernetes

internals.

The decision to use a go-based operator was motivated by its flexibility, allowing for a more compact system.

3.2.2 User-facing API of the SeleniumTestOperator

Every program has a user interface that enables direct communication between the user and the program. An API was explicitly developed for this purpose, allowing users to provide all the data required for test automation. This evolved into the SeleniumTest API and included the SeleniumTest Custom Resource Definition, a corresponding Go object defining it, a reconcile function specifically designed for this custom resource type, and RBAC privileges in cluster roles with bindings.

The custom resource definition of the SeleniumTest is deployed for Kubernetes alongside the operator to enable the user to create SeleniumTest custom resources containing the necessary data to deploy automated tests. As shown in the user manual, the user also needs a test created by the Selenium IDE exported to a .side file, and placed into a config map, to set up a test in the cluster.

By defining custom resources not included in the basic Kubernetes API, Kubernetes Custom Resource Definitions (CRDs) are used to expand the Kubernetes API. Developers can manage essential resources like pods, services, and deployments in the same way that they manage custom resources, thanks to CRDs.

Making a YAML file that details the custom resource's schema is part of defining a CRD. The schema specifies the attributes that can be set and the resource's structure. Additionally, it contains validation rules that guarantee the accuracy of the data entered for the custom resource.

Typically, the CRD YAML file contains the following data:

- Group and version information: Indicates which API group and version the CRD is a part of. In API requests, the CRD is identified by the group and version.
- Kind information: gives specifics about the type of the custom resource. In Kubernetes API queries, the kind is used to specify the kind of the custom resource.

- Metadata: includes details on the CRD, including its name, labels, and annotations.
- Spec: Specifies the fields that can be set and their types, as well as the structure for the custom resource. Additionally, it includes validation rules that guarantee accurate data entry for the custom resource.

Each CRD has a corresponding go object defined within the operator, as well as a reconcile function:

```
1  // SeleniumTestSpec defines the desired state of SeleniumTest
2  type SeleniumTestSpec struct {
3    Repository    string `json:"repository"`
4    Image         string `json:"image"`
5    Tag           string `json:"tag"`
6    Schedule      string `json:"schedule"`
7    ConfigMapName string `json:"configMapName"`
8    Retries       string `json:"retries"`
9    SeleniumGrid  string `json:"seleniumGrid"`
10 }
11 // SeleniumTestStatus defines the observed state of SeleniumTest
12 type SeleniumTestStatus struct {
13   CronJobName string `json:"cronJobName"`
14 }
15 // SeleniumTest is the Schema for the seleniumtests API
16 type SeleniumTest struct {
17   metav1.TypeMeta   `json:",inline"`
18   metav1.ObjectMeta `json:"metadata,omitempty"`
19
20   Spec   SeleniumTestSpec   `json:"spec,omitempty"`
21   Status SeleniumTestStatus `json:"status,omitempty"`
22 }
```

Code 3.1: SeleniumTestResult CR go representation

The reconcile function is the core function responsible for maintaining the desired state of a resource. The reconcile function checks to see if the resource's current state corresponds to the ideal state stated in the resource's Kubernetes manifest.

The Kubernetes API server tells the operator when a change is found in the resource, and the operator then initiates the reconcile function. The reconcile function compares the resource's present state to the desired state listed in the manifest by

reading its current state. The reconcile function takes the necessary steps to resolve discrepancies between the desired and current conditions.

The reconcile function, in general, is a crucial part of a Kubernetes operator and, when appropriately used, guarantees that the resources are always kept in the desired state.

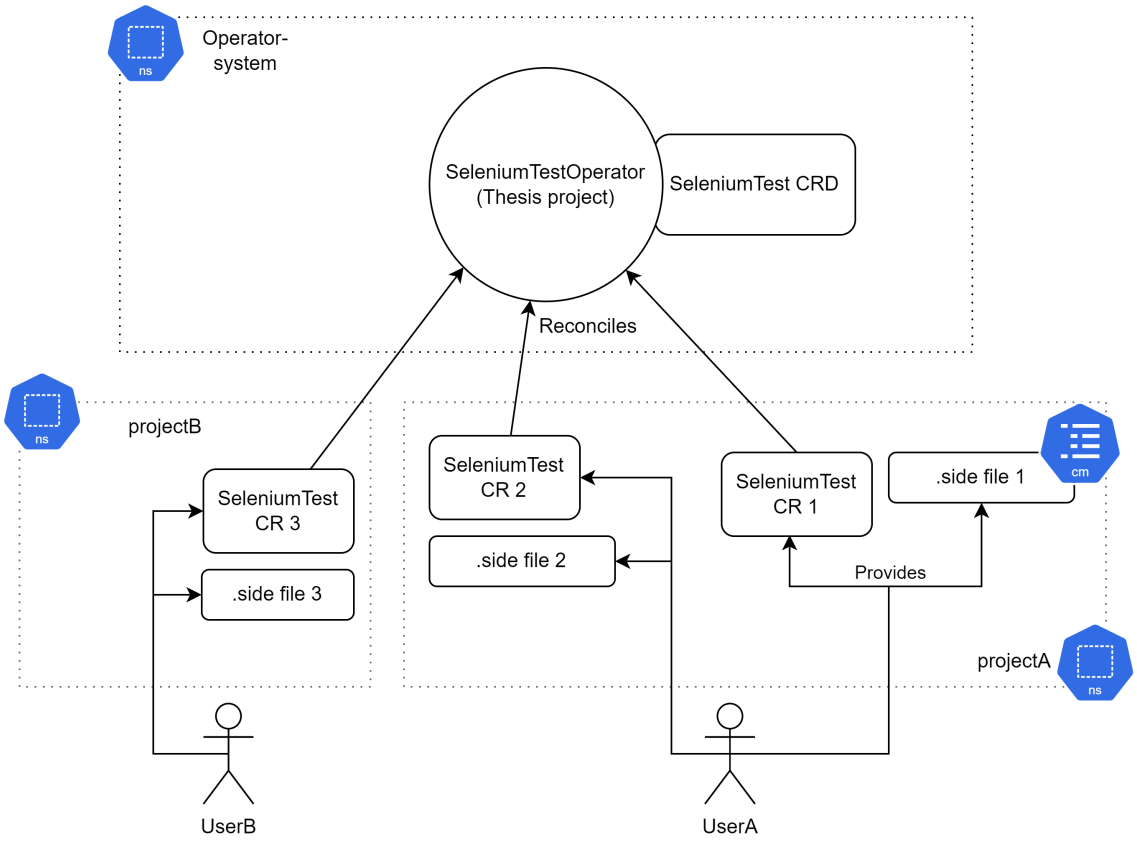

Figure 3.3: Input part of Architecture

3.2.3 Automated scheduling of the tests

The scheduling of the tests is automated by reconciling—the how and why are here two crucial questions. The operator could start each test in a distinct namespace or in the same namespace as the SeleniumTest CR. Building and maintaining a solution with fewer RBAC-related objects and a more straightforward reconcile function would have been simpler if all workloads had been handled in the operator-system namespace. On the other hand, the latter and preferred option offers the opportunity to incorporate the testing workload into tracking distinct namespaces and projects. A desirable non-functional requirement in contemporary business environments is this.

The workload comprises numerous Selenium side runs wrapped in JavaScript code inside a Node.js container. Kubernetes' built-in automation is used for easier maintenance, and the workload is abstracted within pods within jobs within cronjobs.

In order to make sure that a job is consistently created at the appropriate time, the operator creates a cronjob with the specified schedule. The hash code is put after the name of the jobs, which run under the same name as the cronjob.

The test file is accessible within the containers at the "/mnt/config" folder since the operator attached the given configmap as a volume to the job's pod.

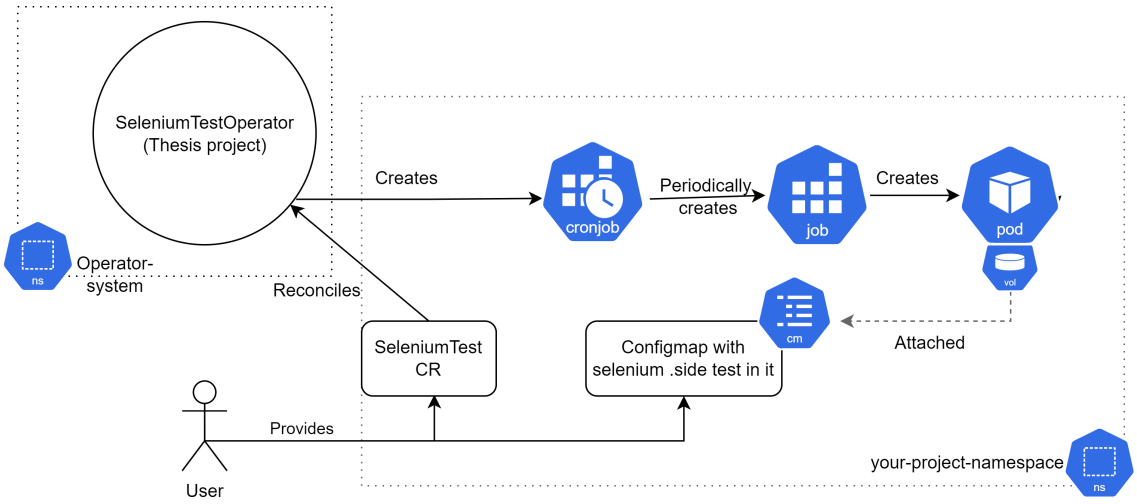

Figure 3.4: Test automation part of Architecture

Without explicitly stating differently, pods use the default service account, severely limiting their capacity to communicate with the cluster. Therefore, the operator has a cluster role with all the permissions the pod's script requires. The operator generates a dedicated service account in the workload namespace for each test and a role binding that links the newly established service account to the previously described cluster role.

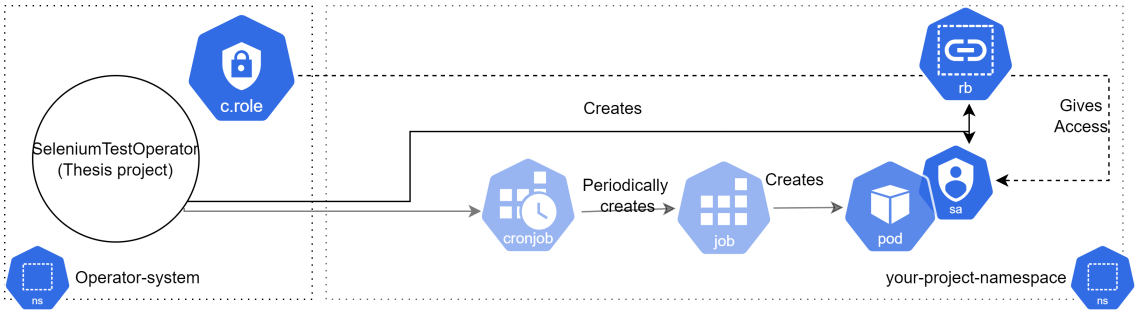

Figure 3.5: RBAC part of Architecture

The SeleniumTest controller's reconcile method, which has the following syntax (logging, exception handling, comments, etc., are not included here for readability in the documentation), handles the construction or validates the existence of all the objects:

```
1  func (r *SeleniumTestReconciler) Reconcile(ctx context.Context,
       req ctrl.Request) (ctrl.Result, error) {
2    instance := &seleniumv1.SeleniumTest{}
3
4    // Validates the ConfigMap with the given name is present
5    err = r.ensureConfigMap(instance)
6    // Checks the ServiceAccount is present, if not, creates it
7    err = r.ensureServiceAccount(instance)
8    // Checks the RoleBinding is present, if not, creates it to
       connect the service account to the cluster role
9    err = r.ensureRoleBinding(instance)
10   // Ensure the CronJob is present, if not, creates it with the
       configmap attached as volume, with the serviceaccount above
11   err = r.ensureCronJob(instance)
12
13   return ctrl.Result{}, nil
14 }
```

Code 3.2: Stripped SeleniumTest reconciler

Each new object (ServiceAccount, RoleBinding, and Cronjob) is also erased along with the relevant SeleniumTest CR to simplify maintenance further. In this manner, the entire lifecycle is managed with only the management of the SeleniumTest CR.

3.2.4 Using Selenium WebDriver for Browser Interactions

Interactions with web browsers are possible using the Selenium WebDriver tool. It offers a collection of procedures and instructions that mimic user operations, including clicking buttons, inputting text, and moving between pages.

A Node.js based container is established with the required Selenium libraries and drivers to use Selenium WebDriver with Node.js with a Selenium Hub deployment (Moon is given as an example in this tutorial, but there are many more possibilities). The container also includes all additional dependencies needed for the tests.

The Selenium WebDriver is used within the container to communicate with the Moon. This entails creating a fresh instance of the appropriate browser, which would then be used to navigate to the desired URL and carry out any necessary interactions described with the .side test files.

Data regarding the test, including page load times, failures, and other performance indicators, is generated while the script runs. Then the test findings are processed and reported to the operator via a custom resource called SeleniumTestResult.

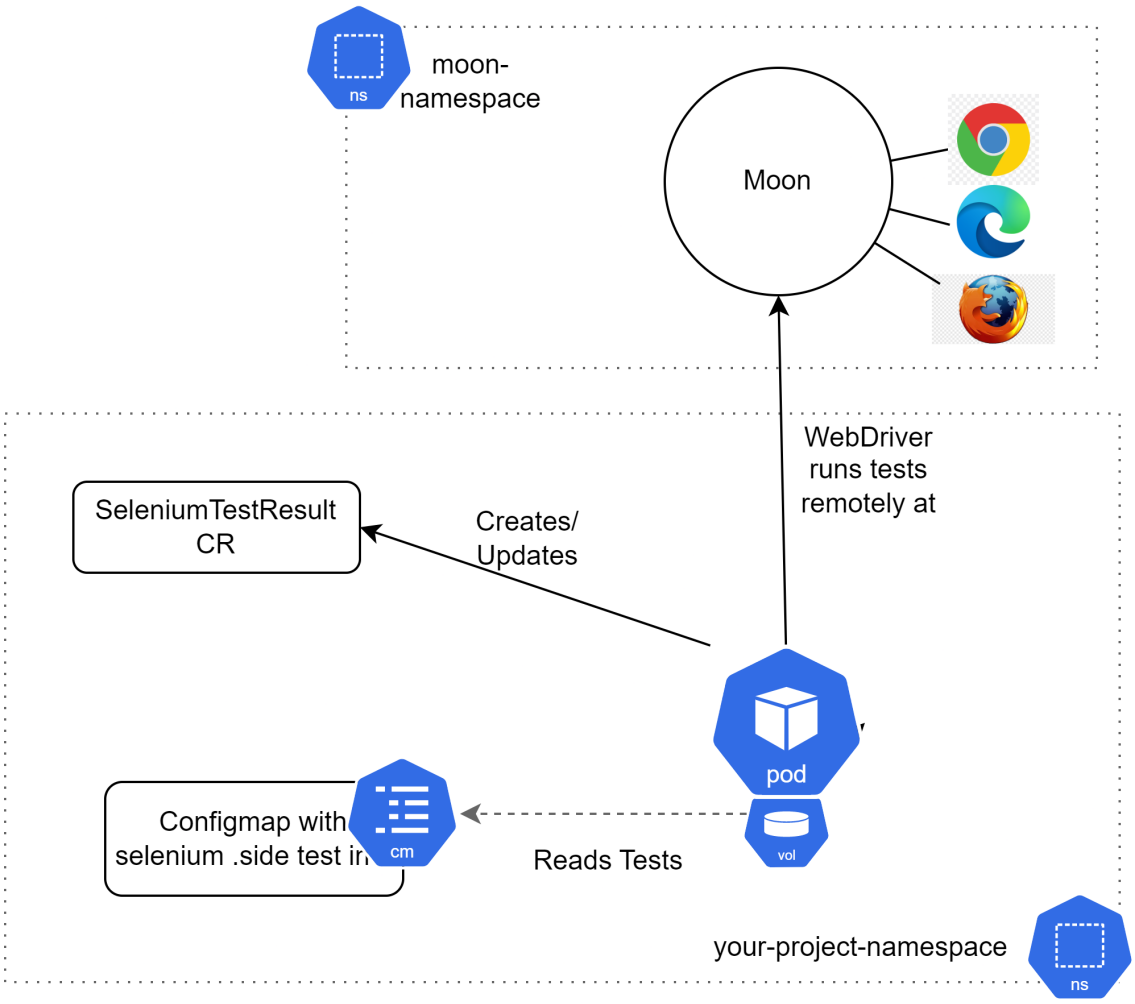

Figure 3.6: SeleniumGrid interaction

3.2.5 Test wrapping container from inside

The operator is constructed around the test runner pod, which serves as the primary function of the objective and is supplemented with features necessary for enterprise integrability and user experience.

The pod is intended to be straightforward, compact, and efficient. In order to initialize and manage a Selenium Webdriver and its communication with the Selenium Grid (moon), it uses Selenium's Selenium-side-runner SDK. The base image of the pod's single container is a Node.js image because the selenium-side-runner can only be used with Node.js. In order to avoid installing any unnecessary compilers on an image that will be run in several instances in a production setting, the wrapper logic was written in JavaScript. The only other package installed is Kubernetes/client-node, which makes it possible to communicate with the Kubernetes API, which is

needed when producing SeleniumTestResult Custom Resources.

The following is the dockerfile defining the container's image:

```
FROM node:18

RUN mkdir /app
WORKDIR /app
RUN apt-get update

RUN npm install @kubernetes/client-node
RUN npm install -g selenium-side-runner
COPY .side.yml .side.yml

COPY runner.js runner.js

ENTRYPOINT ["/bin/sh", "-c" , "node runner.js"]
```

Code 3.3: Dockerfile

The JavaScript code known as runner.js is in charge of running the tests, handling the outcomes, and writing them into a SeleniumTestResult custom resource. The following is how the code would seem without logging, error handling, imports and other utility non-functional requirements.

```
...
execSync('selenium-side-runner -s $SELENIUM_GRID -o results -r
    $RETRIES /mnt/config/*.side');
...
const fileData = fs.readFileSync(path.join('./results', file));
success = fileData.success;
  endTime = fileData.endTime
...
if SeleniumTestResult.isExists():
  updateResult(success, endTime)
else:
  createNewResult(success, endTime);
```

Code 3.4: Simplified runner.js

3.2.6 Evolution of Pod architecture to improve scalability

There are numerous ways to conduct Selenium tests, including using a Selenium grid solution (like moon). This section will go over the various patterns the project used as it progressed from one design to the next and eventually took on the form now shown in the documentation.

First, putting the web driver wrapper and browser runner code in the same container is the most logical approach. This is the simplest method; having both the web driver and the browser on the same local network completely removes any potential networking problems. The drawback is that it can only be as effective as I could make it because the resource-intensive browser process and the lightweight custom runner code are fused. Additionally, the time/resource-saving features Selenium offers cannot be entirely utilized by a standard browser deployment.

The second variation involves dividing the logical elements into two containers within the same pod. This approach enables the operation of browsers with more robustly constructed pre-made images, such as "selenium/standalone-chrome," and is cleaner and simpler to maintain. Since both containers, in this instance, are part of the same pod, they communicate over the same local network. Because it offers advantages without sacrificing anything, this architecture might be regarded as universally superior to the preceding one.

Finally, the entire running logic of the browser can be isolated, becoming a dependency on an external service for the operator. Although networking challenges may exist between the browser and the Selenium web driver, this offers excellent flexibility and may be the most resource-effective alternative. Where the browser workload is processed is up to the user or company. It may be handled within the same cluster (as shown in this documentation), elsewhere, or contracted out to a selenium grid service made available to the general public. The most resource-intensive portion of the testing makes this decision extremely important. This way, some options are publicly competing industry-grade solutions, where the amount of labor involved in optimization is far more tremendous than what the user's team or I can devote. Moon and Selenoid are such examples.

3.2.7 Centralised vs. Decentralised workload management

Workload management also has an architectural decision with varying benefits. Cronjobs, jobs, pods, and SeleniumTestResults might have either been issued in the same namespace as the SeleniumTest CR that initiated them in a decentralized manner or delivered in a centralized manner into the operator-system namespace.

The centralized solution offers the benefit of monitoring the enterprise-level resource costs of selenium live testing. Additionally, it is simpler to implement and requires fewer RBAC-related objects.

The namespaced method, in comparison, has the advantage of making selenium testing and resource monitoring isolation easier. Removing the need for access to a different namespace makes it simpler for the teams to manage the selenium tests for their respective applications.

It is much more critical for businesses to understand a project's cost separately from other projects than to view a part function summed across multiple projects. For this reason, the operator functions in a namespaced architecture.

Although these workloads pale compared to browser automation, the same mental models can be applied. Most industry-standard solutions, like Moon, offer deployment configuration with centralised and namespaced architecture. Moon is deployed centralised in the user manual for ease of demonstration.

3.2.8 SeleniumTestResult CR and exposing results as Prometheus metrics

A SeleniumTestResult custom resource corresponds to each test, and here is where the pod writes the results. In addition to the operator, its CRD is also deployed, and the operator has a go object, a controller, and a distinct reconcile method to handle it.

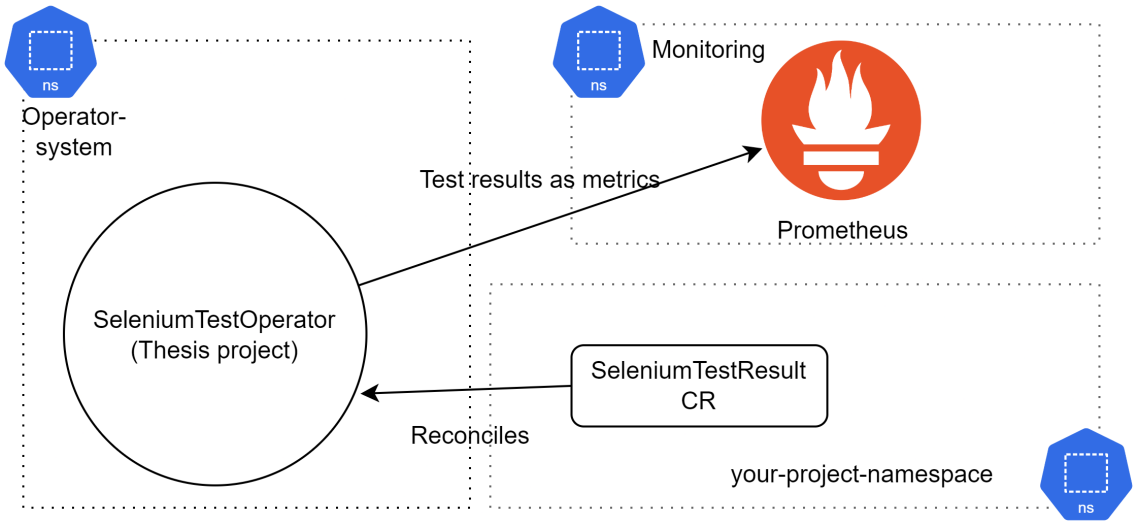

Figure 3.7: Result aggregation part of Architecure

The associated go object:

```
// SeleniumTestResultSpec defines the desired state of
    SeleniumTestResult
type SeleniumTestResultSpec struct {
  Success bool `json:"success"`
  EndTime int  `json:"endTime"`
}
type SeleniumTestResultStatus struct {}

// SeleniumTestResult is the Schema for the seleniumtestresults
    API
type SeleniumTestResult struct {
  metav1.TypeMeta   `json:",inline"`
  metav1.ObjectMeta `json:"metadata,omitempty"`

  Spec   SeleniumTestResultSpec   `json:"spec,omitempty"`
  Status SeleniumTestResultStatus `json:"status,omitempty"`
}
```

Code 3.5: SeleniumTestResult CR go representation

Every time the SeleniumTestResult is modified, the reconcile function is called. The method uses Prometheus labels to expose the data, making it simple to sort the data for test results by name or namespace.

Metric declaration:

```go
var (
  test_results = prometheus.NewGaugeVec(
    prometheus.GaugeOpts{
      Name: "selenium_test_results",
      Help: "0 mean success, 1 means error",
    },
    []string{"test_name", "namespace"},
  )
)
func init() {
  // Register custom metrics with the global prometheus registry
  metrics.Registry.MustRegister(test_results)
}
```

Code 3.6: Custom metric selenium$_{t}est_{r}esultsregistering$

For ease of reading in this documentation, logging and error handling are removed from the reconcile method demonstration below:

```go
func (r *SeleniumTestResultReconciler) Reconcile(ctx context.
    Context, req ctrl.Request) (ctrl.Result, error) {
  instance := &seleniumv1.SeleniumTestResult{}
  labels := prometheus.Labels{"test_name": instance.Name, "
     namespace": instance.Namespace}
  if instance.Spec.Success {
    test_results.With(labels).Set(1)
  } else {
    test_results.With(labels).Set(0)
  }
  return ctrl.Result{}, nil
}
```

Code 3.7: Stripped SeleniumTestResult reconciler

The metric viewed from a prometheus:

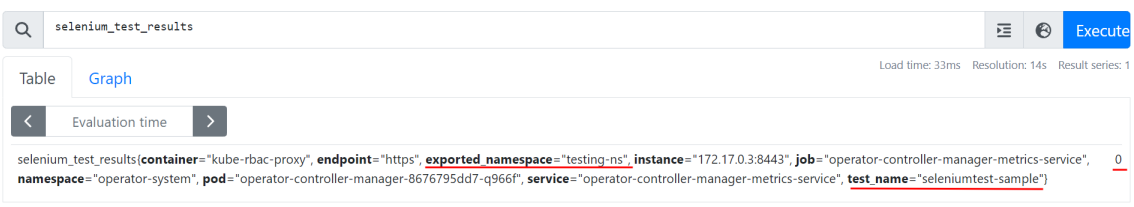

Figure 3.8: Metrics within Prometheus UI

3.3 Testing

A critical component of software development is testing. It confirms that the code operates as planned and complies with all requirements. It is impossible to exaggerate the value of testing because it aids in the early discovery of faults and flaws, lowering total development costs and raising the finished product's caliber.

Here are some details on why code testing is crucial:

- Early Bug Discovery
- Improving Code Quality
- Enhancing User Satisfaction
- Reducing Development Costs
- Boosting Productivity

3.3.1 Unit testing

The software testing that involves isolating specific software application units or components from the rest of the system is known as unit testing. Each code unit is tested separately during unit testing to ensure it operates as intended and satisfies its functional requirements.

Commonly, unit tests concentrate on discrete, small portions of code, such as particular procedures or functions. They are typically automated and frequently run to find and stop bugs as soon as feasible throughout the development process.

The reconciliation logic for the go operator and the JavaCript code for the selenium runner container was developed using unit testing. Without any mocking and utility, the go unit test resembles the following:

```
1  // Objects to track in the fake client.
2  objs := []runtime.Object{seleniumtest}
3
4  s := runtime.NewScheme()
5  assert.NoError(t, scheme.AddToScheme(s))
6  cl := fake.NewFakeClientWithScheme(s)
7
8  // Test when the instance is not found
9  r := &SeleniumTestReconciler{
```

```
    Client: cl,
    Scheme: s,
  }
  req := ctrl.Request{...}
  res, err := r.Reconcile(context.Background(), req)
  assert.NoError(t, err)
  assert.Equal(t, ctrl.Result{}, res)

  // Test when the instance is found
  assert.NoError(t, cl.Create(context.Background(), testInstance))
  res, err = r.Reconcile(context.Background(), req)
  assert.NoError(t, err)
  assert.Equal(t, ctrl.Result{}, res)

  // Test when the ConfigMap already exists
  configMap := &corev1.ConfigMap{...}
  assert.NoError(t, cl.Create(context.Background(), configMap))
  res, err = r.ensureConfigMap(testInstance)
  assert.NoError(t, err)
  assert.Equal(t, nil, res)

  // Test when the ConfigMap does not exist
  assert.NoError(t, cl.Delete(context.Background(), configMap))
  res, err = r.ensureConfigMap(testInstance)
  assert.NoError(t, err)
  assert.Equal(t, true, errors.IsNotFound(err))
```

Code 3.8: Go unit testing

The simplified JavaScript unit test are the following:

```
jest.mock(k8sClient, () => ({
  getNamespacedCustomObject: jest.fn().mockImplementation(arg =>
     ({
    "apiVersion": "selenium.mliviusz.com/v1",
    "kind": "SeleniumTestResult",
    "metadata": {
      "name": "testname"
    },
    "spec": {
      "success": true,
      "endTime": 1678887
```

```
11        }
12      })),
13      createNamespacedCustomObject: jest.fn(),
14    }));
15
16    test("Succesfull CreateNewResult", () => {
17      expect(runner.createNewResult(true, 1435345)).toBe(0);
18    });
19    test("Succesfull UpdateResult", () => {
20      expect(runner.updateResult(true, 1435345)).toBe(0);
21    });
```

Code 3.9: JavaScript unit testing

3.3.2 E2E Integration testing

Two well-liked Golang libraries for creating and running tests are Ginkgo and Gomega. While Gomega is a matcher library that offers a variety of matches to use in tests, Ginkgo is a behavior-driven development (BDD) testing framework that enables writing tests in a more human-readable format. Together, these two libraries may greatly simplify and improve the usability of developing and running tests in Go.

The envtest library is handy when testing Kubernetes applications in Go. In order to test how the application interacts with Kubernetes, Envtest enables spinning up a new Kubernetes control plane inside the test environment. Because it enables testing how code interacts with the Kubernetes API without requiring code to be deployed to a real cluster, this is especially helpful when testing custom controllers or operators.

A test suite is developed that initializes a new Kubernetes control plane using envtest at the start of the test run to leverage Ginkgo, Gomega, and envtest together. Following that, Ginkgo and Gomega are used to write test cases, which involve using application code to create and modify Kubernetes resources and then asserting that the results of those operations match expectations. Finally, the Kubernetes control plane is destroyed using envtest's cleanup functions after the test run. Before deploying the code to a live cluster, this method enables testing the Kubernetes application in a controlled environment to make sure the code functions as planned.

Making e2e tests was made possible by these tools. The tests without the utility functions are as follows:

```
1  func TestAPIs(t *testing.T) {
2    RegisterFailHandler(Fail)
3    RunSpecs(t, "Controller Suite")
4  }
5  var _ = BeforeSuite(func() {
6    By("bootstrapping test environment")
7    cfg, err = testEnv.Start()
8    Expect(err).NotTo(HaveOccurred())
9    Expect(cfg).NotTo(BeNil())
10
11   err = seleniumv1.AddToScheme(scheme.Scheme)
12   Expect(err).NotTo(HaveOccurred())
13   k8sClient, err = client.New(cfg, client.Options{Scheme: scheme.
       Scheme})
14   Expect(err).NotTo(HaveOccurred())
15   Expect(k8sClient).NotTo(BeNil())
16 })
17
18 var _ = AfterSuite(func() {
19   By("tearing down the test environment")
20   err := testEnv.Stop()
21   Expect(err).NotTo(HaveOccurred())
22 })
23
24 var _ = Describe("e2e testing of automating a test", Ordered,
     func() {
25   BeforeAll(func() {
26     By("Installing Moon")
27     Expect(InstallMoon()).To(Succeed())
28   })
29
30   Context("SeleniumTest Operator", func() {
31     It("should run successfully", func() {
32       By("deploying the SeleniumTest operator' controller-manager
           ")
33       var cmd = exec.Command("make", "deploy", fmt.Sprintf("IMG=%
           s", operatorImage))
34       ExpectWithOffset(1, err).NotTo(HaveOccurred())
```

```
35
36          By("validating that the controller-manager pod is running
                as expected")
37            podOutput, err := Run(cmd)
38            ExpectWithOffset(2, err).NotTo(HaveOccurred())
39            podNames := GetNonEmptyLines(string(podOutput))
40            if len(podNames) != 1 {
41              return fmt.Errorf("expect 1 controller pods running,
                    but got %d", len(podNames))
42            }
43            controllerPodName = podNames[0]
44            ExpectWithOffset(2, controllerPodName).Should(
                  ContainSubstring("controller-manager"))
45
46            // Validate pod status
47            status, err := Run(cmd)
48            ExpectWithOffset(2, err).NotTo(HaveOccurred())
49          }
50          EventuallyWithOffset(1, verifyControllerUp, time.Minute,
                time.Second).Should(Succeed())
51
52          By("creating an configmap with a selenium .side test")
53          EventuallyWithOffset(1, func() error {
54            cmd = exec.Command("kubectl", "apply", "-f", filepath.
                    Join(projectDir,
55              "config/samples/sample-configmap.yaml"), "-n",
                    namespace)
56          }, time.Minute, time.Second).Should(Succeed())
57
58          By("creating an instance of the SeleniumTest(CR)")
59          EventuallyWithOffset(1, func() error {
60            cmd = exec.Command("kubectl", "apply", "-f", filepath.
                    Join(projectDir,
61              "config/samples/selenium_v1_seleniumtest.yaml"), "-n",
                    namespace)
62          }, time.Minute, time.Second).Should(Succeed())
63
64          By("validating that the SeleniumTestResult custom resource
                is created or updated")
65          EventuallyWithOffset(1, func() error {
66            cmd = exec.Command("kubectl", "get", "seleniumtestresult"
```

```
            ,
67          "seleniumtest-sample", "-o", "jsonpath={.metadata.
               creationTimestamp}",
68          "-n", namespace,
69        )
70        ExpectWithOffset(2, err).NotTo(HaveOccurred())
71      }, 5*time.Minute, time.Second).Should(Succeed())
72    })
73  })
74  AfterAll(func() {
75    By("Uninstalling Moon")
76    UninstallMoon()
77
78    By("Removing moon namespace")
79    cmd := exec.Command("kubectl", "delete", "ns", "moon")
80    _, _ = Run(cmd)
81  })
```

Code 3.10: E2E testing of operator

Chapter 4

Conclusion

The SeleniumTest Operator is a tool that speeds the testing of web applications. This operator offers a robust and adaptable framework for developing, executing, and monitoring Selenium tests by utilizing the features of Selenium WebDriver, Kubernetes, and Prometheus.

The operator enables developers to quickly design and launch Selenium tests with multiple settings and retrieve comprehensive results, all within a Kubernetes cluster, using custom resources like SeleniumTest and SeleniumTestResult. Integrating with Prometheus makes effective monitoring and alerting based on personalized metrics and thresholds possible.

The Kubernetes API and the usage of containers further assure the portability, scalability, and simplicity of deployment of the SeleniumTest Operator in various scenarios.

Overall, the SeleniumTest Operator represents a considerable improvement in web application testing by giving developers and operators a complete and adaptable solution.

Acknowledgements

I would like to recognize Zvara Zoltán for his significant contribution to the creation of the SeleniumTest operator. His knowledge and commitment were crucial in making this endeavor a success. I am incredibly appreciative of his advice and assistance during the development process. Zvara, I appreciate your sharing your knowledge with me and your support.

Chapter 5

Bibliography

References

- "Go | Operator SDK." Operator SDK, https://sdk.operatorframework.io/docs/building-operators/golang/. Accessed 5 May 2023.
- "Kubernetes." Kubernetes, https://kubernetes.io/. Accessed 5 May 2023.
- "Minikube Start | Minikube." Minikube, https://minikube.sigs.k8s.io/docs/start/. Accessed 5 May 2023.
- prometheus-operator. "GitHub - Prometheus-Operator/Kube-Prometheus: Use Prometheus to Monitor Kubernetes and Applications Running on Kubernetes." GitHub, https://github.com/prometheus-operator/kube-prometheus. Accessed 5 May 2023.
- "Selenium." Selenium, https://www.selenium.dev/. Accessed 5 May 2023.
- Software, Aerokube. "Moon - A Cross Browser Selenium, Cypress, Playwright and Puppeteer Solution for Kubernetes or Openshift Cluster." Aerokube | Efficient Browser Automation Infrastructure, https://aerokube.com/moon/latest/. Accessed 5 May 2023.

List of Figures

List of Codes