Maarten Smeets

Rosanna Denis

**Workshop Graal VM & Quarkus**

# Introduction

The purpose of the workshop is to illustrate some of the powerful features of GraalVM in combination with Quarkus. ‘A Kubernetes Native Java stack tailored for GraalVM & OpenJDK HotSpot, crafted from the best of breed Java libraries and standards’. The workshop will focus on using GraalVM to generate native images, running JVM and native images in Docker containers and deploying to Kubernetes.

This workshop consists of several steps.

**Page 2** Preparations  
Setting up your environment. Import or generate the workshop VM

*Page 7* Create some sample Java applications from the example or create your own

*Page 10* JIT compilation  
Compile your applications “the usual way”

*Page 11* AOT compilation  
Compile into native images

*Page 13* Startup times  
Comparison between the two variants

*Page 14* EXTRA 1.0 - GraalVM and reflection  
Configuration using a tracing agent

*Page 14* EXTRA 1.1 - Native images of Spring Boot applications  
We’re not there yet.. And how to make it work!

*Page 17* Your first Quarkus application  
Importing a Quarkus application in Eclipse and running it in development mode  
Illustrates the live reload functionality

*Page 22* Native image generation  
Generate a native image using a locally installed GraalVM and from a prepared Docker container  
Illustrates start-up time and memory usage differences between JIT and AOT

*Page 25* Docker  
Run Quarkus applications (native and JVM) in a Docker container

*Page 26* EXTRA 2.0 - Quarkus on Kubernetes (bonus)  
Deploy a Quarkus native image Docker container to Minikube  
Generate Kubernetes resources and use them to deploy to Minikube

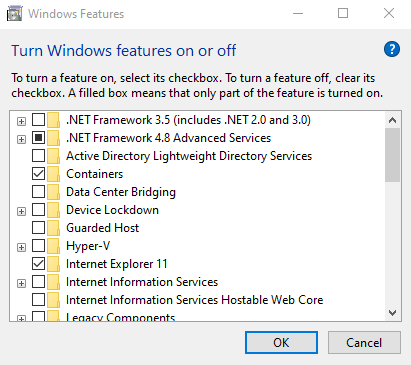
The documents and sources can be found on <https://github.com/AMIS-Services/sig-graalvm-quarkus-15082019>

# Preparations

## Install VirtualBox

It is possible to do the setup of the environment, install GraalVM, Maven, Docker Desktop and the IDE in Windows. However GraalVM on Windows is experimental and there is a (high) probability you won’t be able to get it to generate native images. In order to avoid disappointment, a Linux VM has been prepared in which things do work. This also gives everyone the same environment.

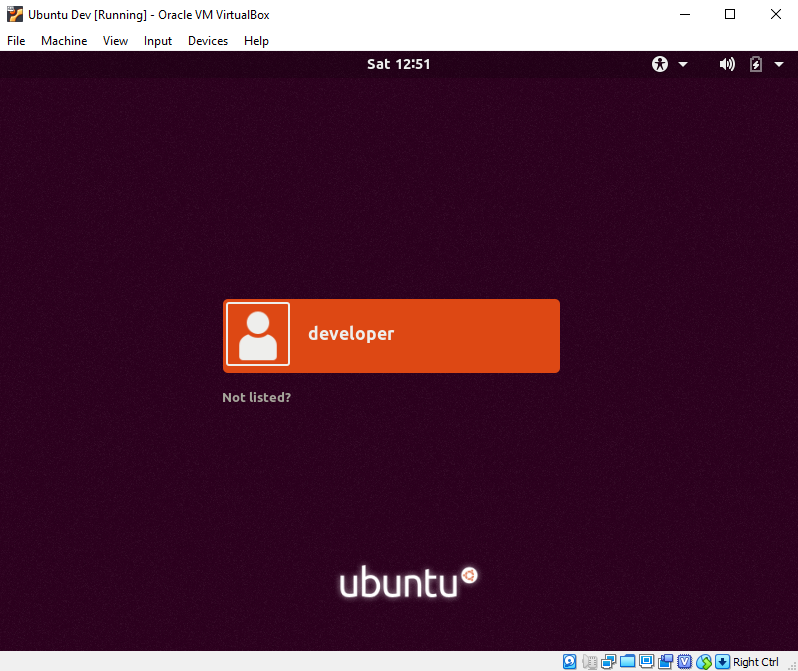
Make sure hardware virtualization support is enabled in the BIOS and make sure Hyper-V is disabled.



Download and install VirtualBox. If you already have Virtualbox installed and are importing the VM, make sure you have at least version 6.0.10 installed.

VirtualBox: <https://www.virtualbox.org/wiki/Downloads>

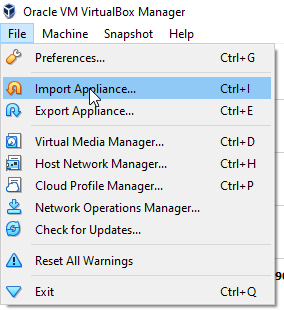
There are two ways to get started with the VM: download it and import in into VirtualBox or build it yourself using Vagrant. The second takes more time. Choose one of the options described below. After you have completed this, you can login using user developer password Welcome01



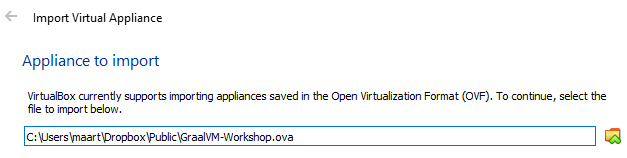
## Download and import the VM in VirtualBox

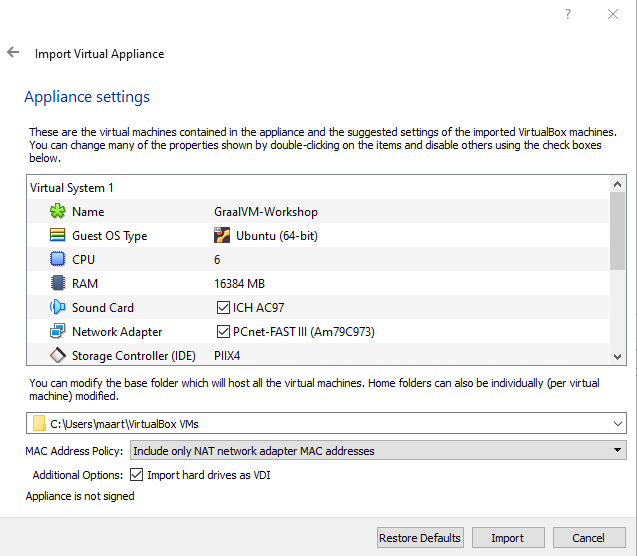
Download the appliance from <https://www.dropbox.com/s/vrecq53mdvi95kc/GraalVM-Workshop.ova?dl=0>

Start VirtualBox and import the appliance.



Select the downloaded OVA file.





Make sure to select the correct MAC address policy else your VM will not have internet access. Also check the available CPUs and RAM of your laptop and adjust the settings accordingly.

Now you can start the imported VM. DO take a look at the provision.sh script in the next section since it shows how to install GraalVM and set it as the default JVM.

## Build the VM yourself using Vagrant

If you have downloaded and imported the VM in the previous step, you should skip this step!

Download and install Vagrant: <https://www.vagrantup.com/>. If you already have Vagrant installed, make sure you have at least version 2.2.5.

Create an empty directory and put the following two files in it.

Vagrantfile   
<https://raw.githubusercontent.com/AMIS-Services/sig-graalvm-quarkus-15082019/master/provision/Vagrantfile>

provision.sh

<https://raw.githubusercontent.com/AMIS-Services/sig-graalvm-quarkus-15082019/master/provision/provision.sh>

The Vagrantfile might have to be updated slightly to reflect the available resources on your laptop (CPUs, memory, disk). The provision.sh script shows the required steps to install GraalVM, Docker, Eclipse, Minikube, Kubectl. As you can see in the provision.sh script, you can use GraalVM like any other JVM.

Start a command prompt in the directory and do:

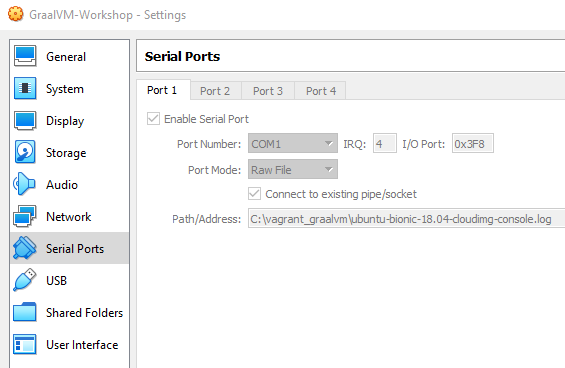
vagrant up

Now you can start the VM

## Troubleshooting

### Serial port

Should the imported or generated VM hang during boot, you might have to configure the Serial port. Make sure a serial port is available and it can send output to a file;



### Internet

Some of the steps require internet availability. One of the issues you might encounter is that the MAC address is hardcoded in the VM in the file /etc/netplan/50-cloud-init.yaml. The MAC might change due to importing the VM. The MAC address listed there should be the same as the output of: cat /sys/class/net/enp0s3/address

In order to fix this, execute the following:

sudo sed -E -i "s/([[:xdigit:]]{1,2}:){5}[[:xdigit:]]{1,2}/`cat /sys/class/net/enp0s3/address`/" /etc/netplan/50-cloud-init.yaml

sudo netplan apply

# Clone the GraalVM\_Quarkus Repo into the VM

Start a terminal, go to your home folder and clone the provided Git repository



cd ~

git clone https://github.com/AMIS-Services/sig-graalvm-quarkus-15082019.git

# Getting Started with java applications

You can find three simple Java applications which are described in this section in ~/sig-graalvm-quarkus-15082019/graalvm

* HelloWorld.java
* TopTen.java
* HelloReflecion.java

***The HelloWorld.java application*** just prints “Hello World” upon execution. It’s a very small and simple application, very suitable for compilation into a native-image.

public class HelloWorld {  
 public static void main(String... args) {  
 System.*out*.println("Hello World");  
 }  
}

***The TopTen.java application*** is a slightly bigger application that is taken from <https://chrisseaton.com/truffleruby/tenthings/>(here they also elaborate on the polyglot functions in GraalVM). The application ploughs through a text “large.txt” and outputs the top 10 most used words in the text. You understand, the bigger the text, the more work it has to process.

import java.io.IOException;  
import java.nio.file.Files;  
import java.nio.file.Paths;  
import java.util.Arrays;  
import java.util.function.Function;  
import java.util.stream.Collectors;  
import java.util.stream.Stream;  
  
public class TopTen {  
  
 public static void main(String[] args) {  
 Arrays.*stream*(args)  
 .flatMap(TopTen::*fileLines*)  
 .flatMap(line -> Arrays.*stream*(line.split("\\b")))  
 .map(word -> word.replaceAll("[^a-zA-Z]", ""))  
 .filter(word -> word.length() > 0)  
 .map(word -> word.toLowerCase())  
 .collect(Collectors.*groupingBy*(Function.*identity*(), Collectors.*counting*()))  
 .entrySet().stream()  
 .sorted((a, b) -> -a.getValue().compareTo(b.getValue()))  
 .limit(10)  
 .forEach(e -> System.*out*.format("%s = %d%n", e.getKey(), e.getValue()));  
 }  
  
 private static Stream<String> fileLines(String path) {  
 try {  
 return Files.*lines*(Paths.*get*(path));  
 } catch (IOException e) {  
 throw new RuntimeException(e);  
 }  
 }  
  
}

***The HelloReflection.java application*** is to demonstrate the use of reflection in a Java application based on <https://medium.com/graalvm/introducing-the-tracing-agent-simplifying-graalvm-native-image-configuration-c3b56c486271>. The main method invokes the included methods when passed on as command line arguments and other arguments result in an exception.

public class HelloReflection {  
 public static void foo() {  
 System.*out*.println("Running foo");  
 }  
 public static void bar() {  
 System.*out*.println("Running bar");  
 }  
 public static void main(String[] args) {  
 for (String arg : args) {  
 try {  
 HelloReflection.class.getMethod(arg).invoke(null);  
 } catch (ReflectiveOperationException ex) {  
 System.*out*.println("Exception running " + arg + ": " + ex.getClass().getSimpleName());  
 }  
 }  
 }  
}

# “Old school” - Just-In-Time Compilation and running on the JVM

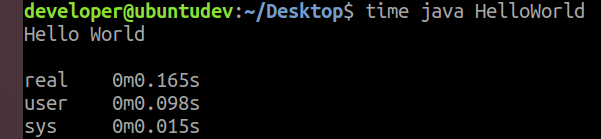
We’ll first do some JIT compilation of all three applications and look at the startup times.

## HElloWorld application

Generate a .class with the javac command and then check the code outputs as expected:

javac HelloWorld.java

time java HelloWorld



## TopTen application

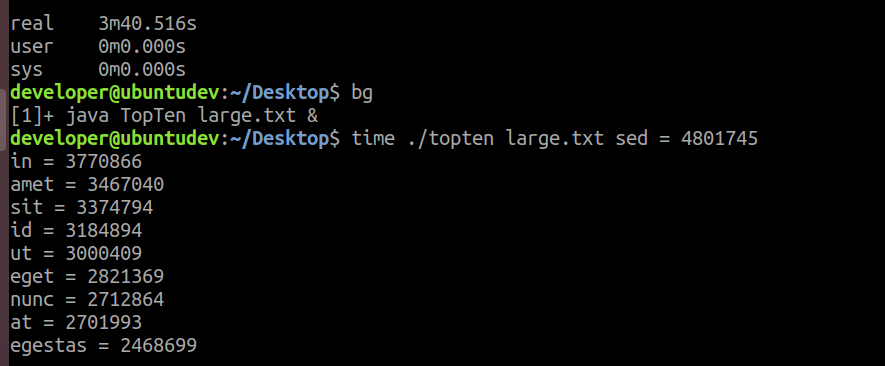
The TopTen application uses large.txt of approximately 800MB. It takes some time for it to process the whole text, so be patient. Unzip it before use:

gunzip large.txt.gz

Compile TopTen by generating a .class file with the javac command:

javac TopTen.java

time java TopTen large.txt

  
As you clearly see, I was not patient enough!

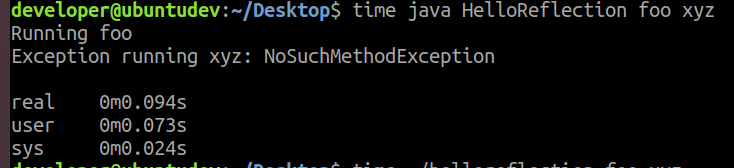
## HElloReflection application

Compile the application that is using reflection with JIT.

javac HelloReflection.java

time java HelloReflection foo xyz

(for printing the error message)



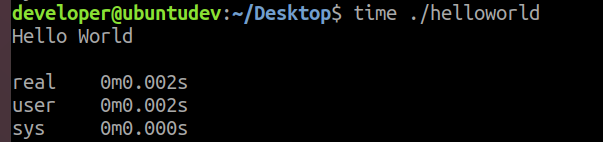
# Graalvm NAtive Image - Ahead-of-Time compilation and standalone execution

Now we’ll use the AOT compilation to generate native-images with GraalVM and then check the time and output of the resulting standalone native executables.

## HElloWorld application

native-image HelloWorld

time ./helloworld



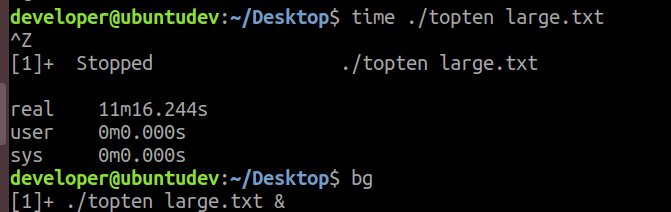
See any differences with the application that’s still running on the JVM? Pretty impressive right? Quite instant.

## TopTen application

No generate a native-image from the TopTen application. Here we’re starting to test the performance of the application. I know it’s probably overkill, but here I used a Lorem Ipsum file of 1.2GB. Just checking. Feel free to use a smaller document! You can generate a lorem ipsum document at <https://loremipsum.io/>. Otherwise use the large.txt in the git repo.

native-image TopTen

time ./topten large.txt

  
See, here I also was not patient enough to wait. It took some 11 minutes and was still not done running.

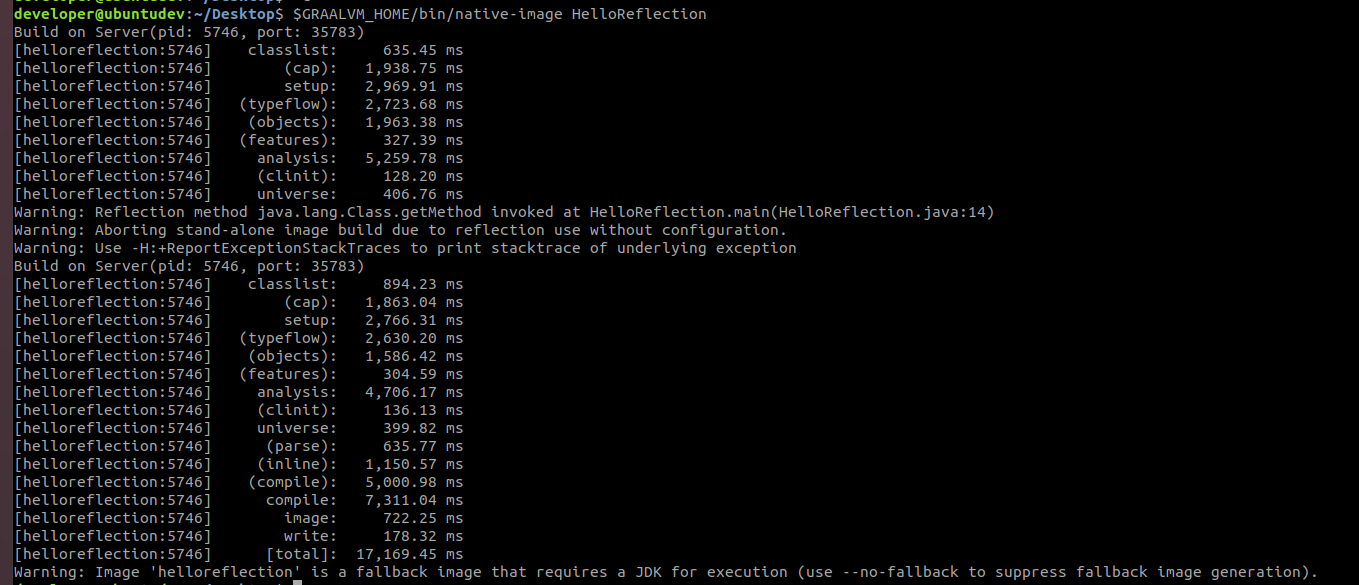
What’s the difference here with the JVM-executed application? Since it’s having a busy job, long run performance starts playing a bigger role. Here you see what its AOT compiling effects are on a longer running process.

## HElloreflection application

And now try creating a native image out of the application using reflection:

native-image HelloReflection

What is going on here?



Apparently GraalVM has a fallback option included and produces an image that is just a launcher for the Java Hotspot VM and wants to keep performing as expected. By trying the AOT compilation again using the flag –-no-fallback, we prevent the fallback image being generated:

native-image -–no-fallback HelloReflection

Now check the functionality of the native image.

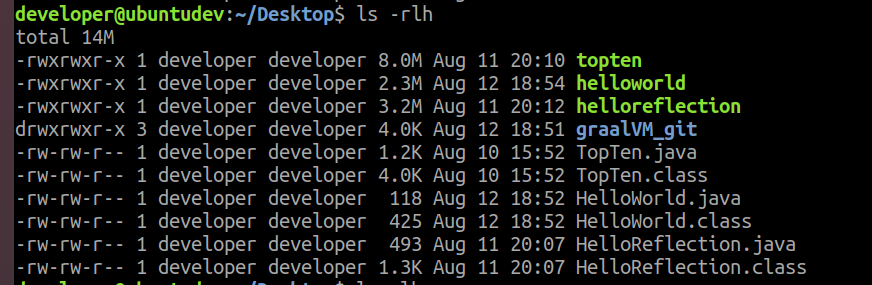
./helloreflection foo bar

What’s the outcome? Did you expect this?  
AOT has some limitations. In particular, the native image executable cannot compile what it doesn’t know about. Individual classes, methods, and fields that should be accessible via reflection need to be known ahead-of-time.

# Size matters – comparing sizes

We’ll check the sizes of the .class files versus the new native images. There are big differences.

$ ls -rlh



What? They’re not smaller than the initial .class files? As mentioned in the presentation, the native image includes the application, the libraries, the JDK and SubstrateVM, explaining the sizes we see here. Sure, for this example 8, 3 and 2mb isn’t that small, but it is **much** smaller than having to package and install an entire JVM (400+mb) including Substrate VM etc! Imagine a large java application in a native image, this might really be beneficially downsized.

# Extra 1.0 – Reflection with configuration by the tracing agent

Only do this part if you feel you have spare time or when you’re interested! Otherwise continue with ‘Your first Quarkus application’.

For making reflection work with GraalVM’s native image, we can use the graalVM tracing agent that comes with it. It makes the method HelloReflection.Foo accessible through reflection:

mkdir -p META-INF/native-image

java -agentlib:native-image-agent=config-output-dir=META-INF/native-image HelloReflection foo xyz

This will create a .JSON file like below, in which the method is specified.

[  
{  
"name":"HelloReflection",  
"methods":[{ "name":"foo", "parameterTypes":[] }]  
}  
]

This way the native image can run as we would expect and then the startup time is massively better 😊

Now you can do:

native-image –no-server -–no-fallback HelloReflection

And run the application

./helloreflection foo bar

Downside is: this is configuration that is cumbersome and time-consuming if you’re doing this for a very large application, since you’d have to include manually all the configurations needed.

# Extra 1.1 – A Spring boot native image??

Only do this part if you feel you have spare time or when you’re interested!

This GraalVM project is under strong development and efforts like Spring (Boot) are trying their best to make it compatible with GraalVM. Spring Framework integrations are in project-phase, where a few projects are in the run to develop Spring and Spring Boot compatibility for/with GraalVM. For now we would still bump into all kinds of problems trying to generate a native-image from these popular applications.

Here an example of what happens when you want to native-image a Spring Boot jar.

***Simple HelloWorld application:*** Create a simple springboot application which implements CommandLineRunner and build it with Maven into a .jar.

package com.graalvm.helloworld;  
  
import org.springframework.boot.CommandLineRunner;  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
  
@SpringBootApplication  
public class HelloworldApplication implements CommandLineRunner {  
  
 public static void main(String[] args) {  
 SpringApplication.*run*(HelloworldApplication.class, args);  
 }  
  
 public void run(String... args) {  
 System.*out*.println("Printing \n \n Hello World");  
 }  
  
}

time java -jar helloworld-0.0.1-SNAPSHOT.jar

. \_\_\_\_ \_ \_\_ \_ \_

/\\ / \_\_\_'\_ \_\_ \_ \_(\_)\_ \_\_ \_\_ \_ \ \ \ \

( ( )\\_\_\_ | '\_ | '\_| | '\_ \/ \_` | \ \ \ \

\\/ \_\_\_)| |\_)| | | | | || (\_| | ) ) ) )

' |\_\_\_\_| .\_\_|\_| |\_|\_| |\_\\_\_, | / / / /

=========|\_|==============|\_\_\_/=/\_/\_/\_/

:: Spring Boot :: (v2.1.7.RELEASE)

2019-08-10 16:30:08.337 INFO 4552 --- [ main] c.g.helloworld.HelloworldApplication : Starting HelloworldApplication v0.0.1-SNAPSHOT on AMIS-Rosanna with PID 4552 (C:\Users\Rosanna\OneDrive - Conclusion\Documenten\Academy\SIG\_GraalVM\_Quarkus\helloworld\target\helloworld-0.0.1-SNAPSHOT.jar started by Rosanna in C:\Users\Rosanna\OneDrive - Conclusion\Documenten\Academy\SIG\_GraalVM\_Quarkus\helloworld\target)

2019-08-10 16:30:08.351 INFO 4552 --- [ main] c.g.helloworld.HelloworldApplication : No active profile set, falling back to default profiles: default

2019-08-10 16:30:09.060 INFO 4552 --- [ main] c.g.helloworld.HelloworldApplication : Started HelloworldApplication in 1.244 seconds (JVM running for 1.811)

Printing

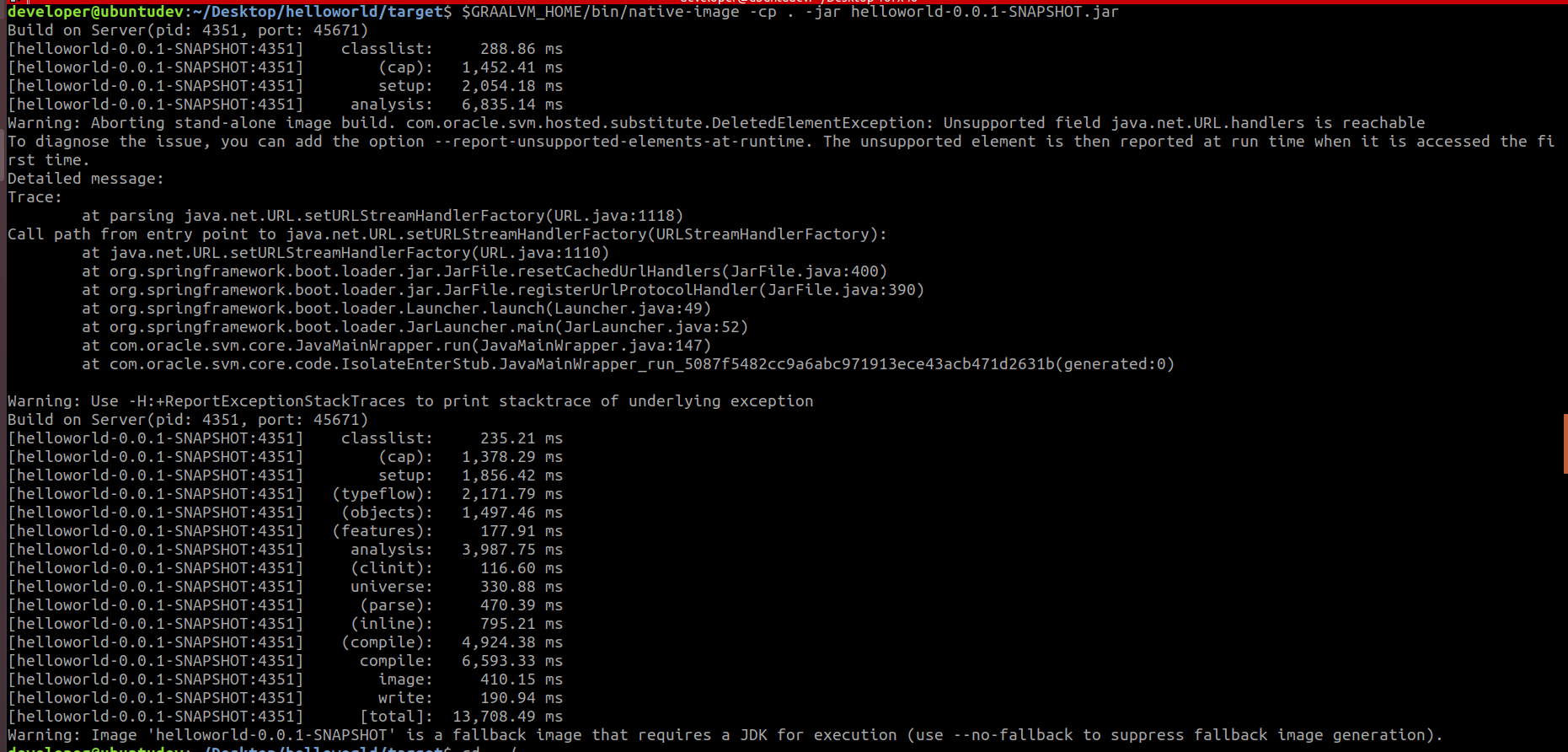
Hello World

real 0m1.916s

user 0m0.015s

sys 0m0.000s

native-image -cp . -jar helloworld-0.0.1-SNAPSHOT.jar



OOPSIE… Can’t process that! It has some pretty bad troubles with the start up of spring boot (heavily relying on reflection?). So for getting things to work, you’d need some help of the holy grail of Open Source: <https://github.com/spring-projects/spring-framework/wiki/GraalVM-native-image-support> for example. They’re very passionate.

Also, many projects are coming in to save the day. For example, Micronaut has built some stuff around the startup of Spring Boot to get it passed through to the native image generator of GraalVM. <https://github.com/micronaut-projects/micronaut-spring/tree/master/examples/greeting-service>

# Demo-time

# Your first Quarkus application

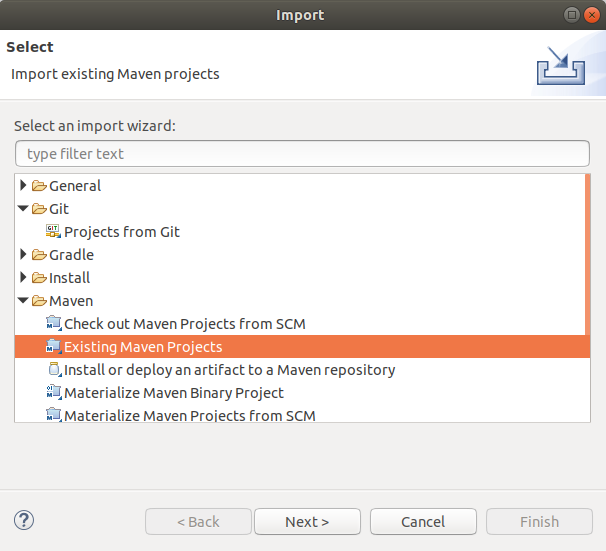
This includes the quarkus-getting-started project which can be created by following the following tutorial: <https://quarkus.io/guides/getting-started-guide>. Read this tutorial and understand what has been prepared. We’ll use it as a starting point.

Here, we’re going to get the Quarkus project started and running.

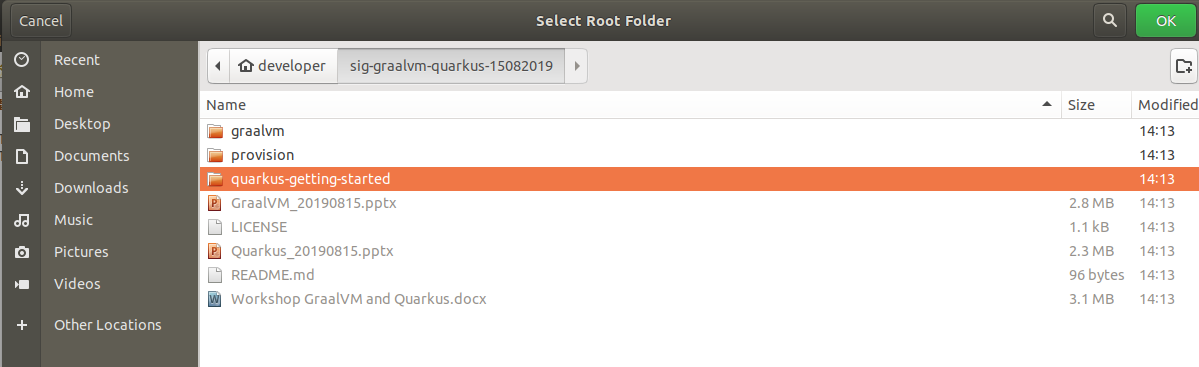
Start Eclipse

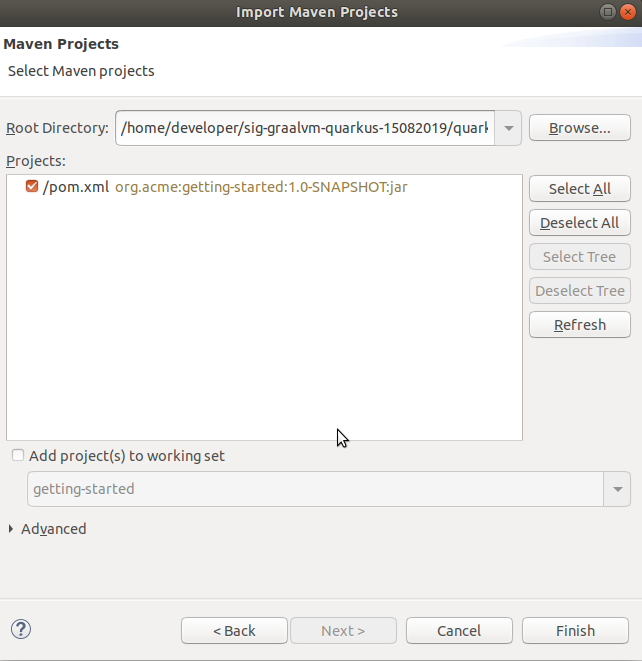


File, Import, Existing Maven projects

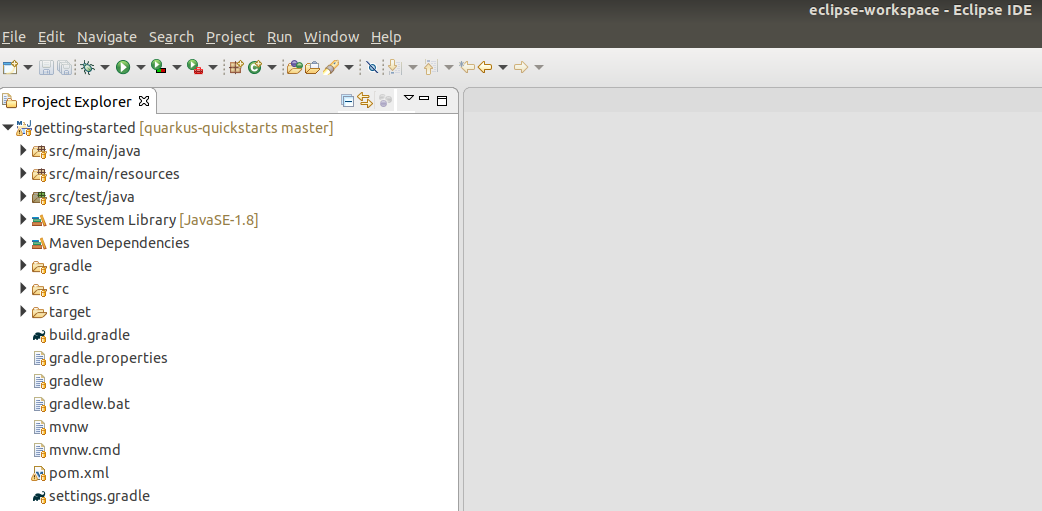


Select the quarkus-getting-started folder

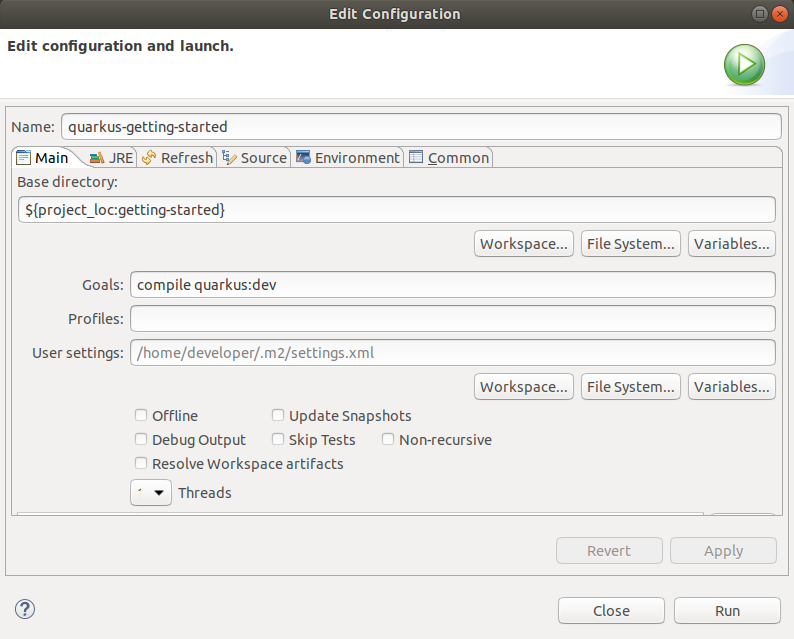




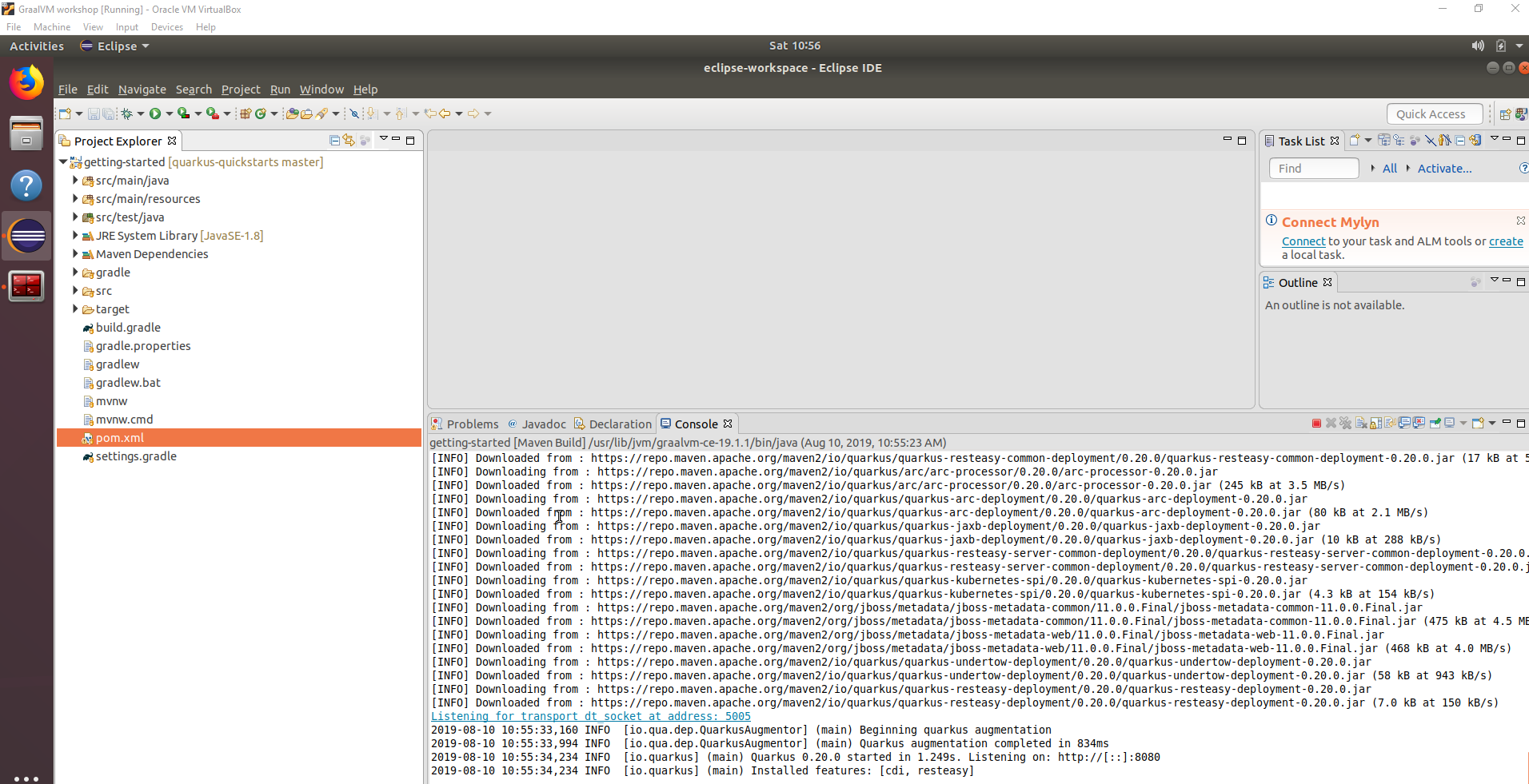
Open the Project explorer (Window, Show view, Project explorer) or use the Package explorer



Right click the pom.xml, Run As, Maven build …



Click Run and confirm the application is running:



In a terminal give the following command to make it return “Hello”:

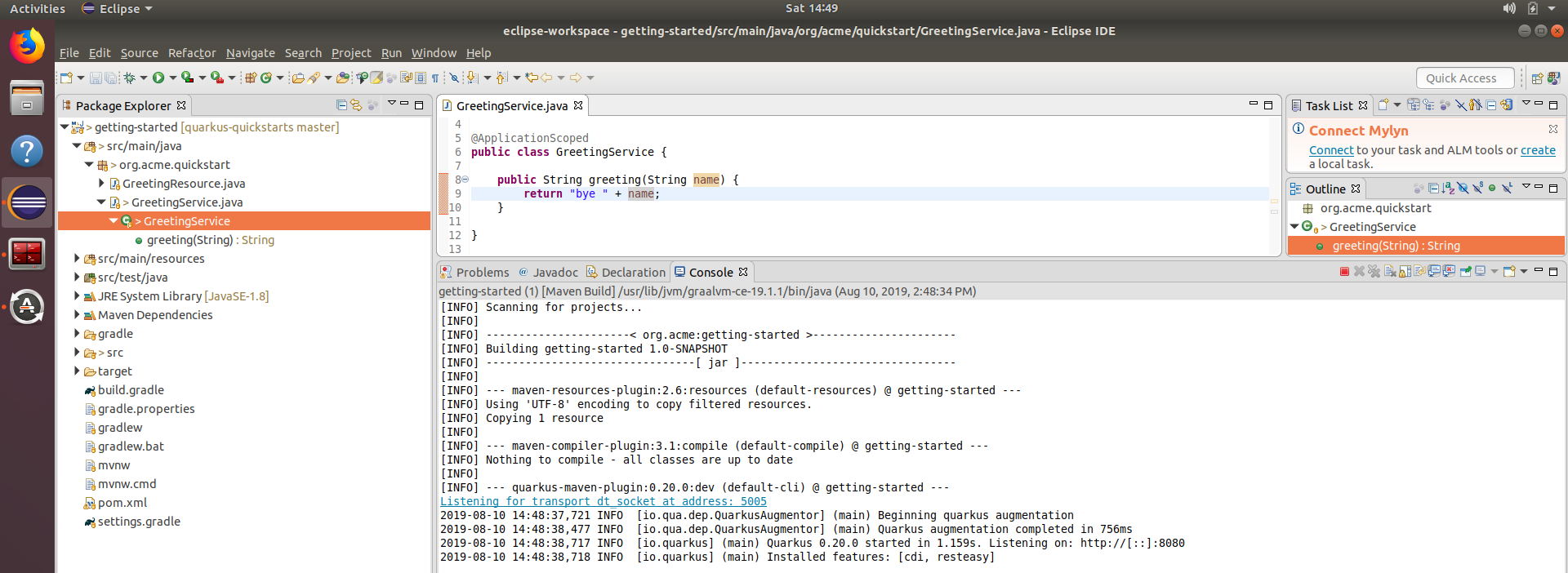
curl http://localhost:8080/hello

Confirm it returns hello

## Live reload

Now we’re testing the live reload function with Quarkus, which is very developer friendly!

Update the GreetingService.java and save



Notice nothing peculiar happens next to the file being saved.

Now fire off a new request

curl http://localhost:8080/hello

and check the response. Check the Console in Eclipse. What has happened?

Notice that when you do a mvn package, 2 JAR files get generated.



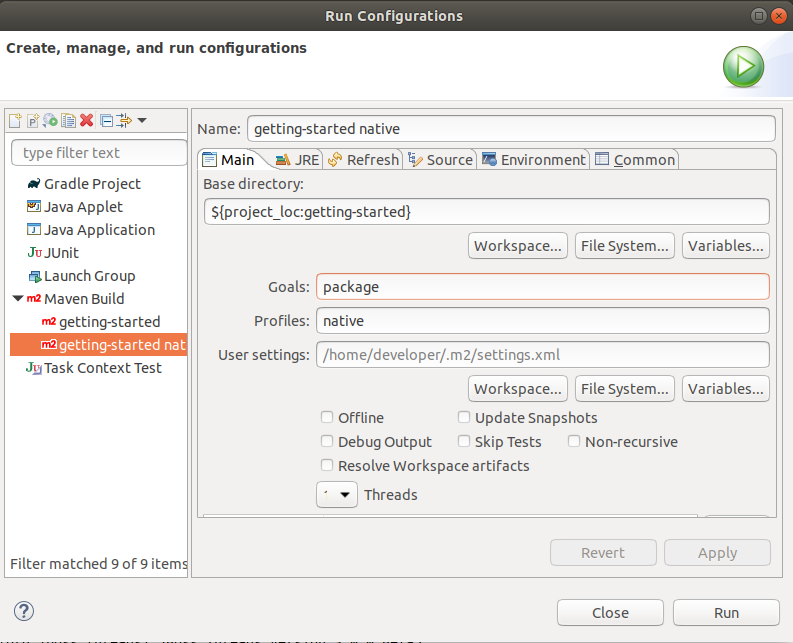
The runner has dependencies in the target/lib folder. This is the file which can be run with java -jar. The smaller JAR file contains the actual application logic. Both are not suitable to be compiled to a native image since they are not a single ‘fat jar’ containing everything which is required.

Restore the GreetingService to returning hello again or update the tests in order to avoid errors during the next steps.

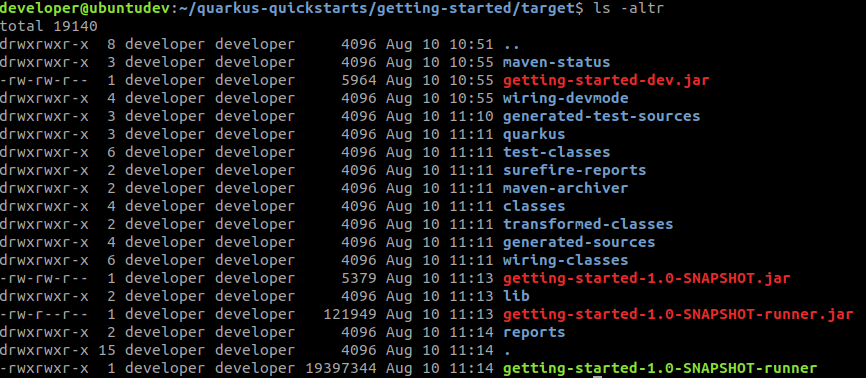
# Native image generation

Now the fun part: we’re generating a native image with our Quarkus project.

Create a new Maven run configuration



Confirm the native image has been created



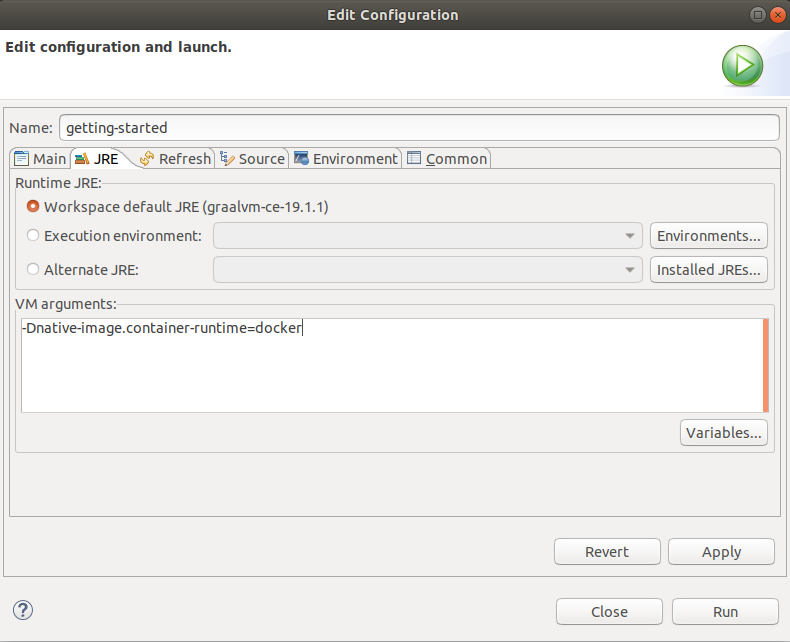
Execute it



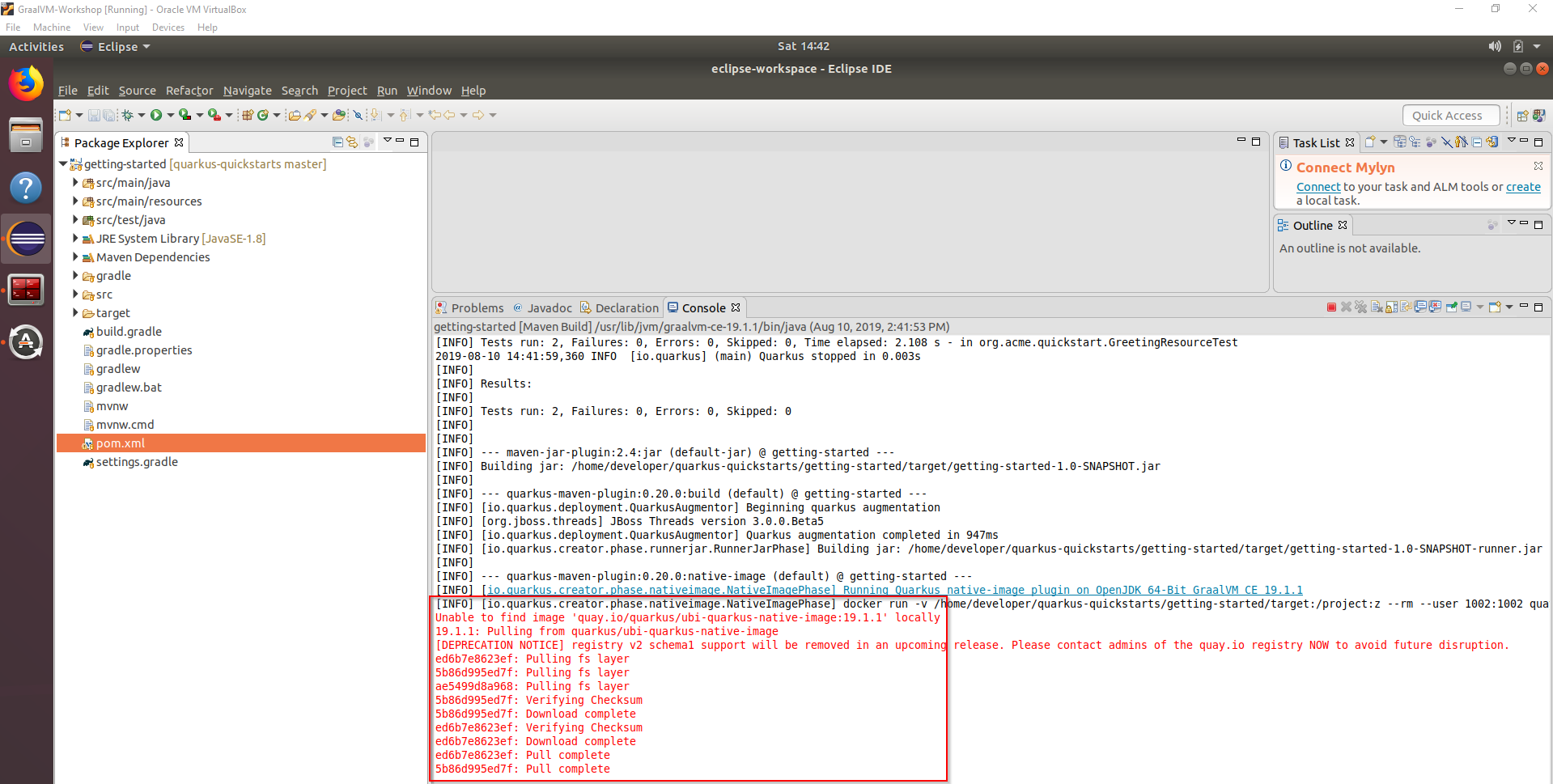
Compare the start-up time of the native executable with the start-up time of the JAR. If you do not have the JAR running anymore, rerun it. Otherwise check the screenshots in this workshop.

## Build the native image without locally installed GraalVM

Add the following JVM argument and rerun the build



What is happening here?



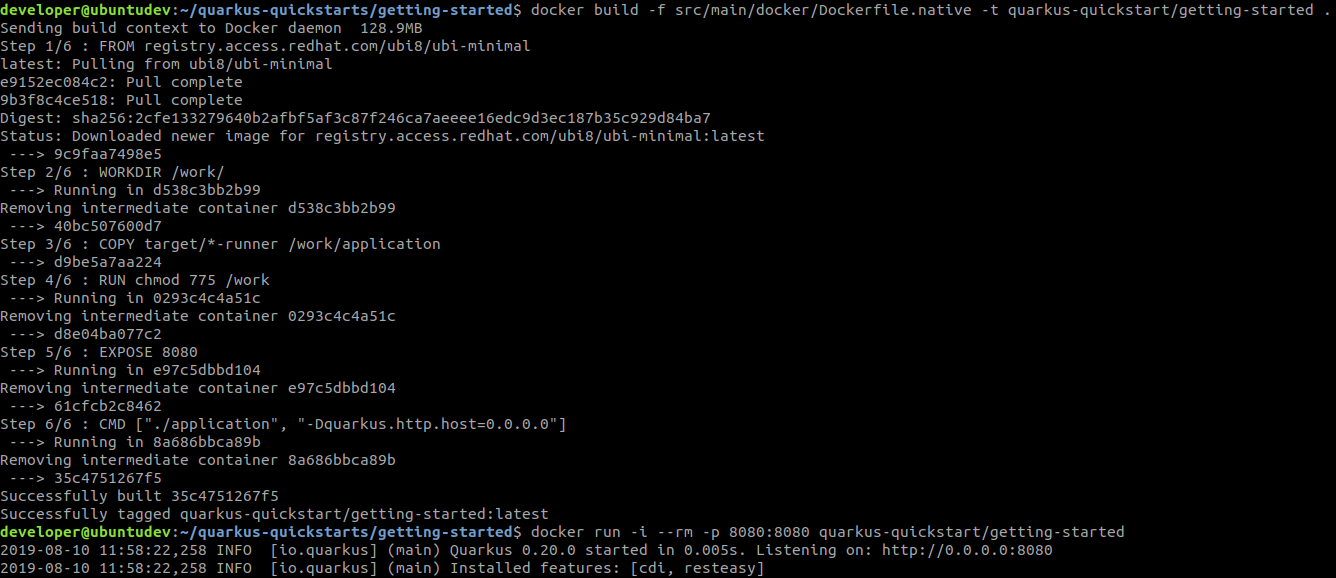
# Docker

Build and run the native image in a Docker container

cd /home/developer/sig-graalvm-quarkus-15082019/quarkus-getting-started

docker build -f src/main/docker/Dockerfile.native -t quarkus-quickstart/getting-started .

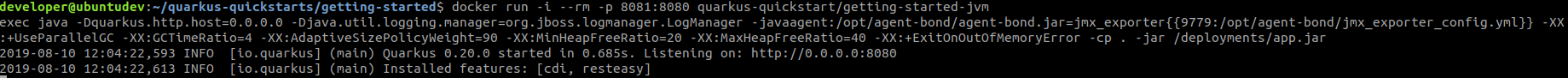
docker run -i --rm -p 8080:8080 quarkus-quickstart/getting-started



Build and run the JVM image in a Docker container

docker build -f src/main/docker/Dockerfile.jvm -t quarkus-quickstart/getting-started-jvm .

docker run -i --rm -p 8081:8080 quarkus-quickstart/getting-started-jvm



Compare the memory usage of the two containers

docker stats



## Cleanup

docker stop $(docker ps -a -q)

# extra 2.0 - Quarkus on Kubernetes (Bonus)

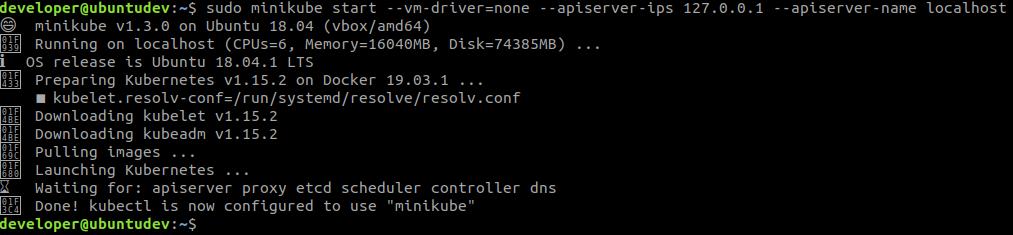
This part is extra. Only do this part if you feel you have spare time or when you’re interested! Excuse us, we’re just enthousiastic 😉

## Start minikube

sudo minikube start --vm-driver=none --apiserver-ips 127.0.0.1 --apiserver-name localhost

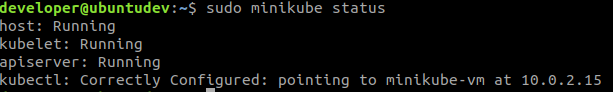
sudo chown -R developer /home/developer/.minikube

sudo chgrp -R developer /home/developer/.minikube



sudo minikube update-context

sudo minikube status

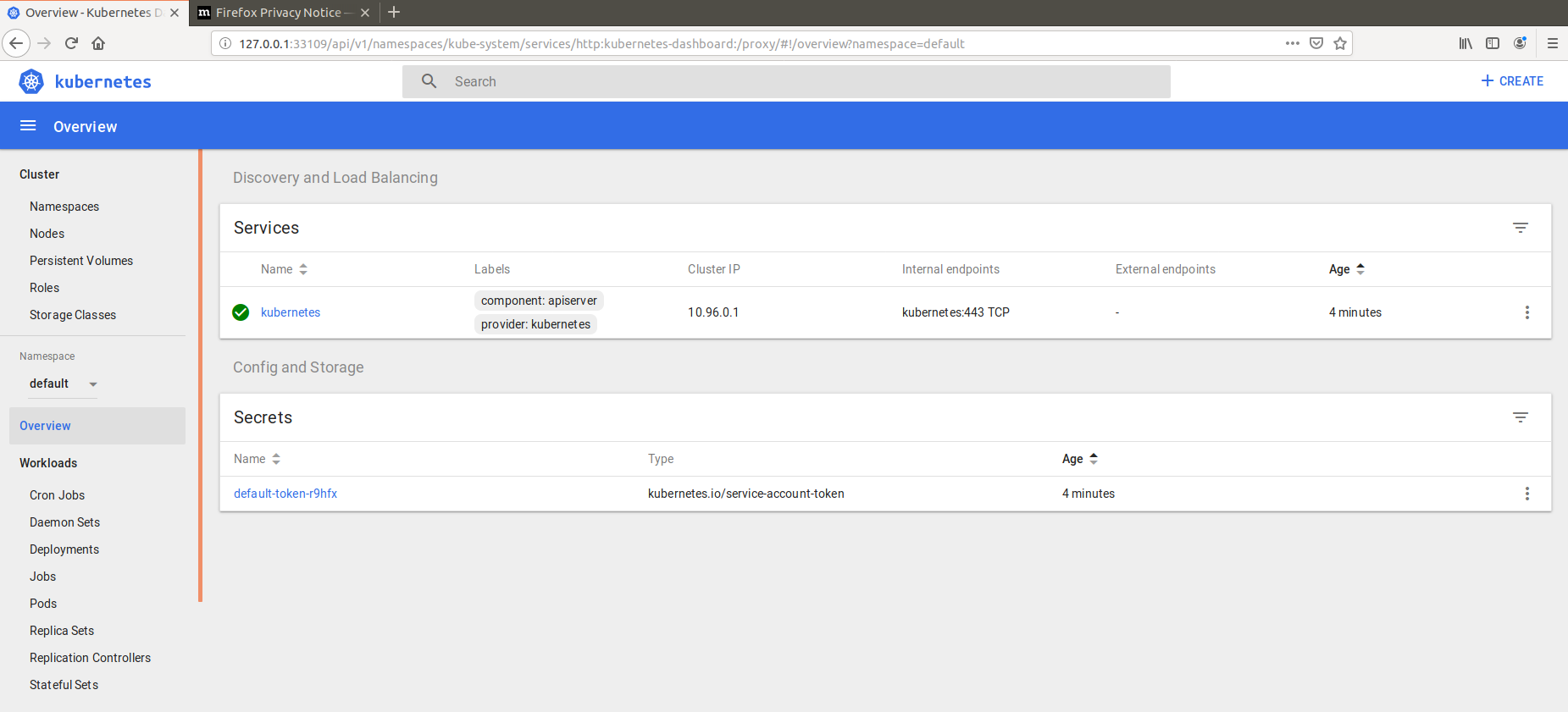


sudo minikube dashboard

This displays an URL.



Open this URL in a browser to display the Minikube dashboard



## Deploy the container to Kubernetes

Based on <https://blog.hasura.io/sharing-a-local-registry-for-minikube-37c7240d0615/>

Download and create a Minikube hosted Docker registry and expose it

wget https://gist.githubusercontent.com/coco98/b750b3debc6d517308596c248daf3bb1/raw/6efc11eb8c2dce167ba0a5e557833cc4ff38fa7c/kube-registry.yaml -O kube-registry.yaml

sudo kubectl create -f kube-registry.yaml

Change the push repository of the container

docker tag quarkus-quickstart/getting-started localhost:5000/quarkus-quickstart/getting-started

Push to the locally hosted registry (so Minikube can find it)

docker push localhost:5000/quarkus-quickstart/getting-started

Start the container

sudo kubectl run quarkus-quickstart --image=localhost:5000/quarkus-quickstart/getting-started --port=8080 --image-pull-policy=IfNotPresent

Expose the container

sudo kubectl expose deployment quarkus-quickstart --type=NodePort

Test the deployment

curl $(sudo minikube service quarkus-quickstart --url)/hello

## Generating and using Kubernetes resources (Extra)

Update the getting-started project with the Kubernetes extension. Generate resources and use these to deploy to the running Minikube instance.

Get inspiration from the following and the above example on how to do this: <https://quarkus.io/guides/kubernetes-resources>