Continuous Monitoring on Docker with ELK Stack

By,

Sreehari C

sreeharichakkavalapil@gmail.com

Github Link

https://github.com/sreeharichak/Phase5CMonDockerwithELKstack.git

Project Objective:

Continuous Monitoring (CM) on Docker with ELK stack is the process that helps developers to monitor the application in real-time using Kibana.

Background of the problem statement:

XYZ Technology Solutions hired you as a DevOps Engineer. The company is undergoing an infrastructural change regarding the tools used in the organization. The company decides to implement DevOps to develop and deliver the products. Since XYZ is an agile organization, they follow Scrum methodology to develop the projects incrementally. They decide to dockerize their applications so that they can deploy them on Kubernetes. Each application when deployed and exposed, will have a unique URL and port, using which we can access that application.

Requirement Analysis:

The application and its versions should be available on GitHub
- Commit the code multiple times and track their versions on GitHub
- Build the application in Docker, and host it in Docker Hub
- Deploy ELK stack on Docker and push application logs to it
- Automate Docker build and deployment using Jenkins pipeline code

<u>Tools used:</u>

- Docker
- Docker Compose
- Elasticsearch
- Logstash
- Kibana
- Spring Boot application

Complete Implementation steps along with detailed description

I Docker

In this project, following steps are followed

1. I**nstall Docker container in windows**, to **create docker image**
2. D**eploy Docker image** (which as one simple spring boot based microservice) in developer machine.

Docker Installation

To install docker on Windows 10 machine, follow the below steps:

Choose the appropriate Docker installer for your System

Before starting with the installation process, we need to understand the exact Docker version that is suitable for the Windows that you are using. Docker has provided two versions of Windows distribution as bellow

- For windows 10 we need to follow this link https://docs.docker.com/docker-for-windows/

We will follow the Docker toolbox installation steps

Download Docker installer

We need to first download the Docker toolbox distribution from https://download.docker.com/win/stable/DockerToolbox.exe and we will follow the installation steps in local Workstation.

Enable Hardware Virtualization Technology

In order for the Docker toolbox to work properly we need to make sure your Windows system supports Hardware Virtualization Technology and that virtualization is enabled. Docker has provided detailed step on this here: https://docs.docker.com/toolbox/toolbox_install_windows/#step-1-check-your-version.

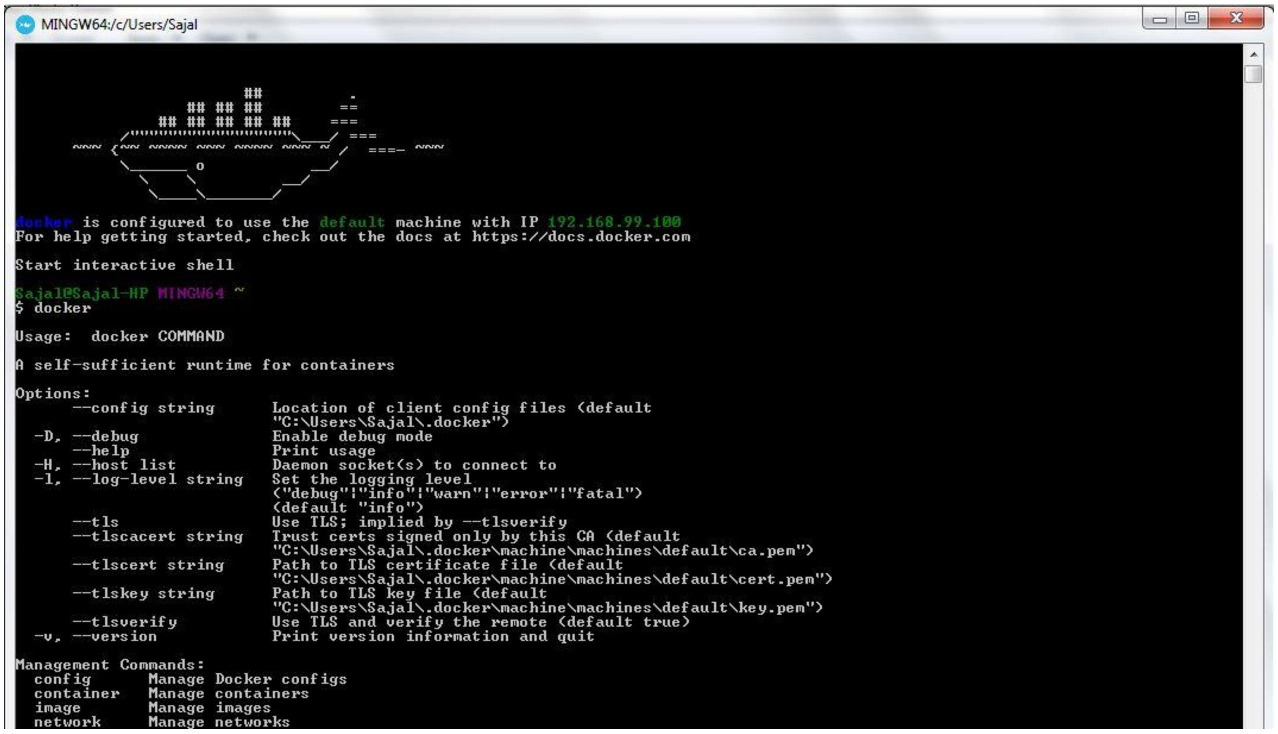
If we don’t have this enabled then we need to go to the BIOS option and enable Hardware Virtualization. The BIOS is a bit different for different models of Computer, so please follow the official guideline for enabling that.

Run Docker installer

Once we have the Installer downloaded and we have enabled the Hardware Virtualization, we can start the installer. It’s just like another windows based installation process guided by an installation wizard.

Verify your installation

To verify docker installation, open *Docker QuickStart Terminal* shortcut from either Desktop or Start menu. Verify that the Docker prompt is coming and then need to test a few basic commands. Docker prompt and sample docker command will look like below.

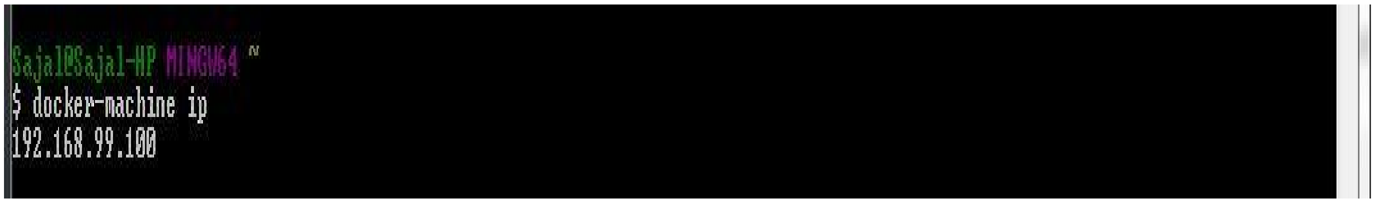


```
MINGW64:/c/Users/Sajal

docker is configured to use the default machine with IP 192.168.99.100
For help getting started, check out the docs at https://docs.docker.com

Start interactive shell

Sajal@Sajal-HP MINGW64 ~
$ docker

Usage:  docker COMMAND

A self-sufficient runtime for containers

Options:
      --config string      Location of client config files (default
                           "C:\Users\Sajal\.docker")
  -D, --debug              Enable debug mode
      --help               Print usage
  -H, --host list          Daemon socket(s) to connect to
  -l, --log-level string   Set the logging level
                           ("debug"|"info"|"warn"|"error"|"fatal")
                           (default "info")
      --tls                Use TLS; implied by --tlsverify
      --tlscacert string   Trust certs signed only by this CA (default
                           "C:\Users\Sajal\.docker\machine\machines\default\ca.pem")
      --tlscert string     Path to TLS certificate file (default
                           "C:\Users\Sajal\.docker\machine\machines\default\cert.pem")
      --tlskey string      Path to TLS key file (default
                           "C:\Users\Sajal\.docker\machine\machines\default\key.pem")
      --tlsverify          Use TLS and verify the remote (default true)
  -v, --version            Print version information and quit

Management Commands:
  config      Manage Docker configs
  container   Manage containers
  image       Manage images
  network     Manage networks
```

Docker installation verification

Note Down the Docker IP

We need to now note down the Docker IP assigned to this Container. We will access this IP to access the Applications installed inside Docker. To know the IP from the command prompt use command docker-machine ip. Here is the sample output of the command. Please note that this IP will be different for different M/Cs.



```
Sajal@Sajal-HP MINGW64 ~
$ docker-machine ip
192.168.99.100
```

docker-machine ip output

Create docker Image

We will first create a spring boot based REST API, add docker specific configuration and then we will create docker image.

Create Spring REST Project

Develop Microservice for testing. I have used spring boot and Maven and Eclipse as IDE. Add and REST endpoints so that once this application is deployed into Docker, we can test this by accessing the rest endpoint.

```
package com.example.howtodoinjava.hellodocker;

import java.util.Date;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@SpringBootApplication
public class HelloDockerApplication {

    public static void main(String[] args) {
        SpringApplication.run(HelloDockerApplication.class, args);
    }
}
```

```
@RestController
class HelloDockerRestController {
    @RequestMapping("/hello/{name}")
    public String helloDocker(@PathVariable(value = "name") String name) {
        String response = "Hello " + name + " Response received on : " + new Date();
        System.out.println(response);
        return response;

    }
}
```

Update resources/application.properties with server port information.

```
server.port = 9080
```

Now test this microservice by running the project as a spring boot application.

Add Docker Configurations

Now create a file named Dockerfile in the root directory and add the below lines as Docker configurations.

```
FROM openjdk:8-jdk-alpine
VOLUME /tmp
ADD target/hello-docker-0.0.1-SNAPSHOT.jar hello-docker-app.jar
ENV JAVA_OPTS=""
ENTRYPOINT [ "sh", "-c", "java $JAVA_OPTS -Djava.security.egd=file:/dev/./urandom -jar /h
docker-app.jar" ]
```

This is used by Docker while creating the image. It is basically declaring the Java runtime information and target distributions.

Add Maven Docker Plugins

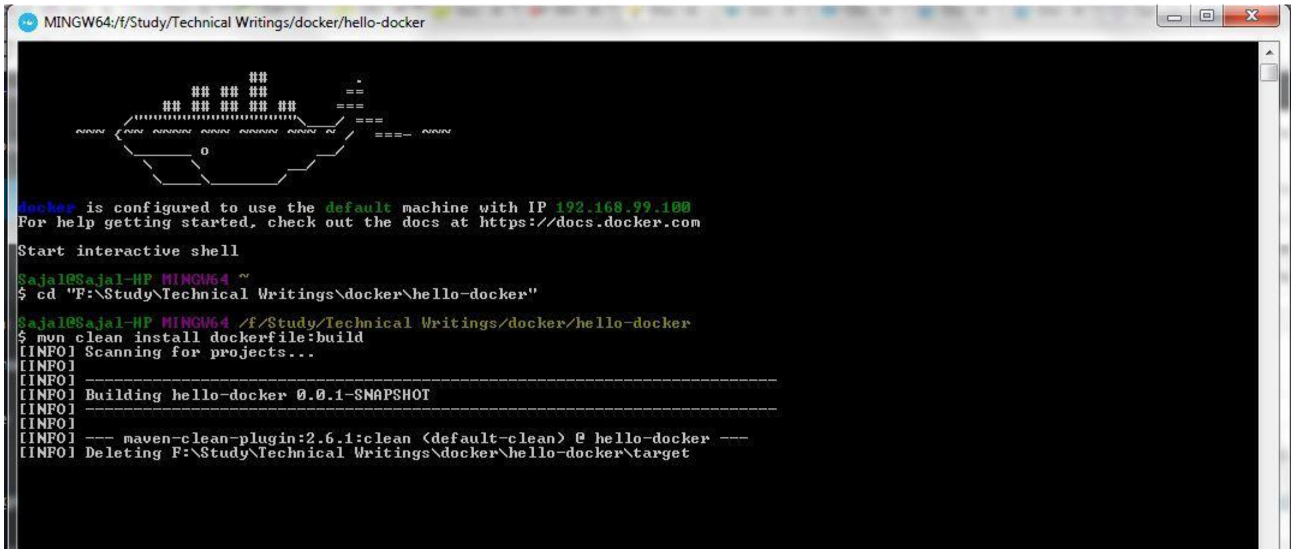
Add two maven plugins in the pom.xml file so that we can use the Docker related maven commands while creating the instance. Those plugins are dockerfile-maven-plugin and maven-dependency-plugin.

We have used the minimal configurations required to build the project.

```
<plugin>
    <groupId>com.spotify</groupId>
    <artifactId>dockerfile-maven-plugin</artifactId>
    <version>1.3.4</version>
    <configuration>
        <repository>${docker.image.prefix}/${project.artifactId}</repository>
    </configuration>
</plugin>
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-dependency-plugin</artifactId>
    <executions>
        <execution>
            <id>unpack</id>
            <phase>package</phase>
            <goals>
                <goal>unpack</goal>
            </goals>
            <configuration>
                <artifactItems>
                    <artifactItem>
                        <groupId>${project.groupId}</groupId>
                        <artifactId>${project.artifactId}</artifactId>
                        <version>${project.version}</version>
                    </artifactItem>
                </artifactItems>
            </configuration>
        </execution>
    </executions>
</plugin>
```

Create Docker Image

Now use maven command `mvn clean install dockerfile:build` to create docker image.



```
MINGW64:/f/Study/Technical Writings/docker/hello-docker

                        ##         .
                  ## ## ##        ==
               ## ## ## ## ##    ===
           /"""""""""""""""""\___/ ===
      ~~~ {~~ ~~~~ ~~~ ~~~~ ~~~ ~ /  ===- ~~~
           \______ o           __/
             \    \         __/
              \____\_______/

docker is configured to use the default machine with IP 192.168.99.100
For help getting started, check out the docs at https://docs.docker.com

Start interactive shell

Sajal@Sajal-HP MINGW64 ~
$ cd "F:\Study\Technical Writings\docker\hello-docker"

Sajal@Sajal-HP MINGW64 /f/Study/Technical Writings/docker/hello-docker
$ mvn clean install dockerfile:build
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building hello-docker 0.0.1-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- maven-clean-plugin:2.6.1:clean (default-clean) @ hello-docker ---
[INFO] Deleting F:\Study\Technical Writings\docker\hello-docker\target
```

Docker Image build from Docker terminal

Please make sure your local application is not running while you are building the image, in that case you might get maven build failure, as in clean step it will not be able to delete the target folder as the jar is being used by java process

Here is the last few lines of the maven output log where it is building the image.

```
[INFO] Image will be built as hello-howtodoinjava/hello-docker:latest
[INFO]
[INFO] Step 1/5 : FROM openjdk:8-jdk-alpine
[INFO] Pulling from library/openjdk
[INFO] Digest: sha256:2b1f15e04904dd44a2667a07e34c628ac4b239f92f413b587538f801a0a57
[INFO] Status: Image is up to date for openjdk:8-jdk-alpine
[INFO]  ---> 478bf389b75b
[INFO] Step 2/5 : VOLUME /tmp
[INFO]  ---> Using cache
[INFO]  ---> f4f6473b3c25
[INFO] Step 3/5 : ADD target/hello-docker-0.0.1-SNAPSHOT.jar hello-docker-app.jar
[INFO]  ---> ce7491518508
[INFO] Removing intermediate container c74867501651
[INFO] Step 4/5 : ENV JAVA_OPTS ""
[INFO]  ---> Running in f7cd27710bf3
[INFO]  ---> 086226135205
[INFO] Removing intermediate container f7cd27710bf3
[INFO] Step 5/5 : ENTRYPOINT sh -c java $JAVA_OPTS -Djava.security.egd=file:/dev/./uran
docker-app.jar
[INFO]  ---> Running in 9ef14a442715
```

```
[INFO]  ---> bf14919a32e2
[INFO] Removing intermediate container 9ef14a442715
[INFO] Successfully built bf14919a32e2
[INFO] Successfully tagged hello-howtodoinjava/hello-docker:latest
[INFO]
[INFO] Detected build of image with id bf14919a32e2
[INFO] Building jar: F:\Study\Technical Writings\docker\hello-docker\target\hello-docker-0.0.1
docker-info.jar
[INFO] Successfully built hello-howtodoinjava/hello-docker:latest
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
```

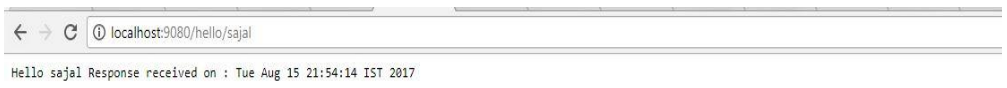
Deploy and Run Docker Image

So we have created the Docker Image (i.e. hello-docker-0.0.1-SNAPSHOT-docker-info.jar). We also have an installed docker container running on our local machine.

Now, to run the docker image inside the installed docker container, we will use the below command.

```
docker run -p 8080:9080 -t hello-howtodoinjava/hello-docker  --name hello-docker-image
```

Here the option `-p 8080:9080` is important. It says that expose port `8080` for internal port `9080`. Remember our application is running in port `9080` inside docker image and we will access that in port `8080` from outside Docker container.
Now access the application with URL http://192.168.99.100:8080/hello/sajal.
Notice that the browser output is same as output of standalone REST API on `localhost`.

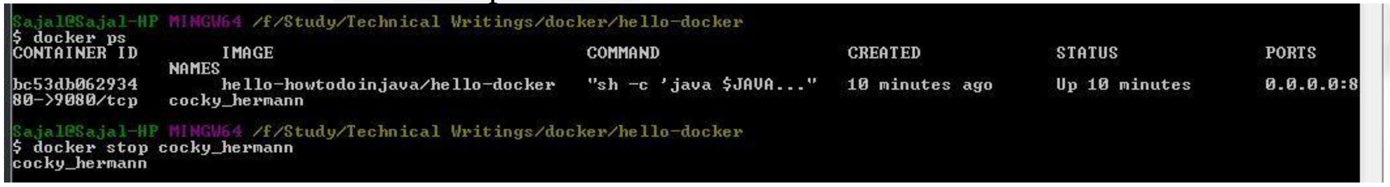


```
localhost:9080/hello/sajal
Hello sajal Response received on : Tue Aug 15 21:54:14 IST 2017
```

Docker Localhost Output

Stop Docker Container

We can list down all docker containers by command docker ps in the terminal and we can use command docker stop <name>



```
Sajal@Sajal-HP MINGW64 /f/Study/Technical Writings/docker/hello-docker
$ docker ps
CONTAINER ID        IMAGE                               COMMAND                  CREATED             STATUS              PORTS
                    NAMES
bc53db062934        hello-howtodoinjava/hello-docker    "sh -c 'java $JAVA..."   10 minutes ago      Up 10 minutes       0.0.0.0:8
80->9080/tcp        cocky_hermann

Sajal@Sajal-HP MINGW64 /f/Study/Technical Writings/docker/hello-docker
$ docker stop cocky_hermann
cocky_hermann
```

Stop Docker Container

The project deployed successfully on docker

II Integrating ELK stack to micro services ecosystem.

Elastic search, Logstash and Kibana– together referred as **ELK stack**. They are used for searching, analyzing, and visualizing log data in real time.

Steps followed

1. What is ELK Stack

1. **Elasticsearch** is a distributed, JSON-based *search and analytics engine* designed for horizontal scalability, maximum reliability, and easy management.

2. **Logstash** is a dynamic *data collection pipeline* with an extensible plugin ecosystem and strong Elasticsearch synergy.
3. **Kibana** gives the *visualization* of data through a UI.

2. ELK stack configuration

All these three tools are based on JVM and before starting installing them, please verify that JDK has been properly configured. Check that standard JDK 1.8 installation, `JAVA_HOME` and `PATH` setup is already done.

2.1. Elasticsearch

- Download latest version of Elasticsearch from this https://www.elastic.co/downloads/elasticsearch and unzip it any folder.
- Run `bin\elasticsearch.bat` from command prompt.
- By default, it would start at http://localhost:9200

2.2. Kibana

- Download the latest distribution from https://www.elastic.co/downloads/kibana and unzip into any folder.
- Open `config/kibana.yml` in an editor and set `elasticsearch.url` to point at your Elasticsearch instance. In our case as we will use the local instance just uncomment `elasticsearch.url: "http://localhost:9200"`
- Run `bin\kibana.bat` from command prompt.
- Once started successfully, Kibana will start on default port `5601` and Kibana UI will be available at http://localhost:5601

2.3. Logstash

- Download the latest distribution from https://www.elastic.co/downloads/logstash and unzip into any folder.
- Create one file `logstash.conf` as per configuration instructions. The exact configuration is shown below.

  Now run `bin/logstash -f logstash.conf` to start logstash

ELK stack is not up and running. Now we need to create a few microservices and point logstash to the API log path.

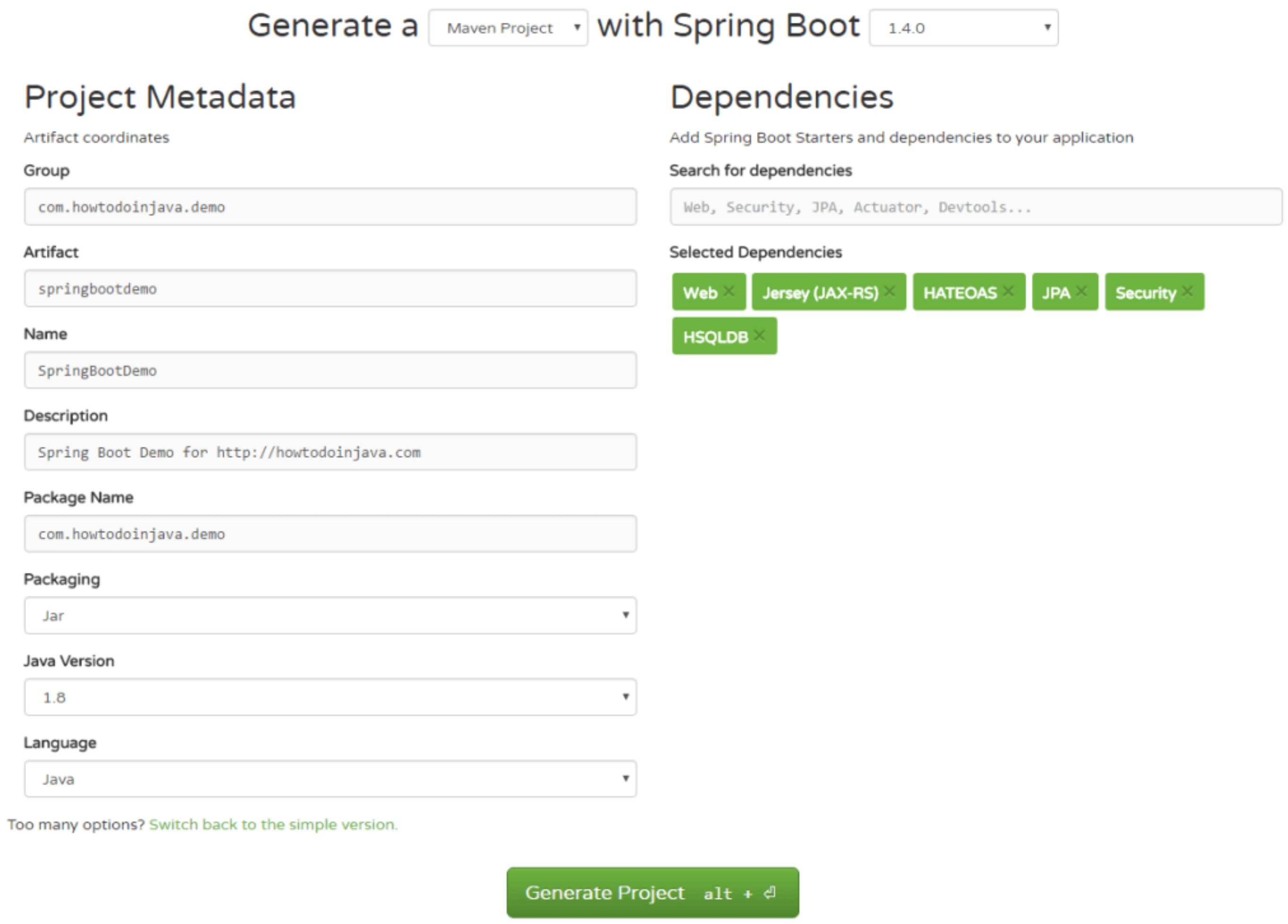
3. ELK stack – Create Microservice

3.1. Create Spring Boot Project

Crate an application using spring boot for faster development time. Follow those steps to start this service.

3.1. Create spring boot hello world project template

To create a template for spring boot application, use http://start.spring.io/. Here, you can select all dependencies which you have currently in mind, and generate the project.



Generate a Maven Project with Spring Boot 1.4.0

Project Metadata

Artifact coordinates

Group: com.howtodoinjava.demo

Artifact: springbootdemo

Name: SpringBootDemo

Description: Spring Boot Demo for http://howtodoinjava.com

Package Name: com.howtodoinjava.demo

Packaging: Jar

Java Version: 1.8

Language: Java

Too many options? Switch back to the simple version.

Dependencies

Add Spring Boot Starters and dependencies to your application

Search for dependencies: Web, Security, JPA, Actuator, Devtools...

Selected Dependencies: Web, Jersey (JAX-RS), HATEOAS, JPA, Security, HSQLDB

Spring Boot Options

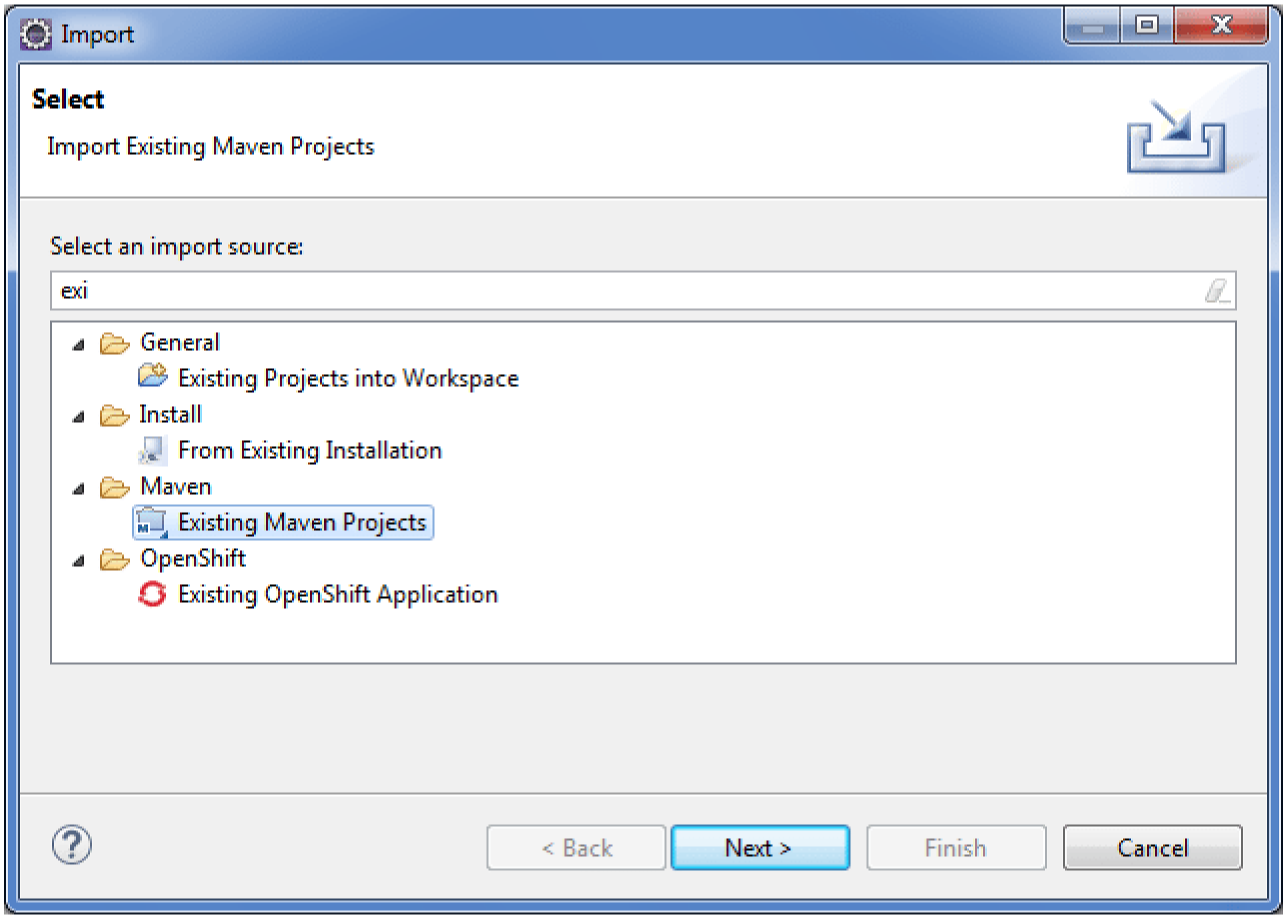
I have selected dependencies like Jersey, Spring,web , Spring HATEOAS, Spring JPA and Spring Security etc. You can add more dependencies after you have downloaded and imported the project or in future when requirements arise.

Generate Project button will generate a .zip file. Download and extract the file into your workspace.

2. Import spring boot project to eclipse

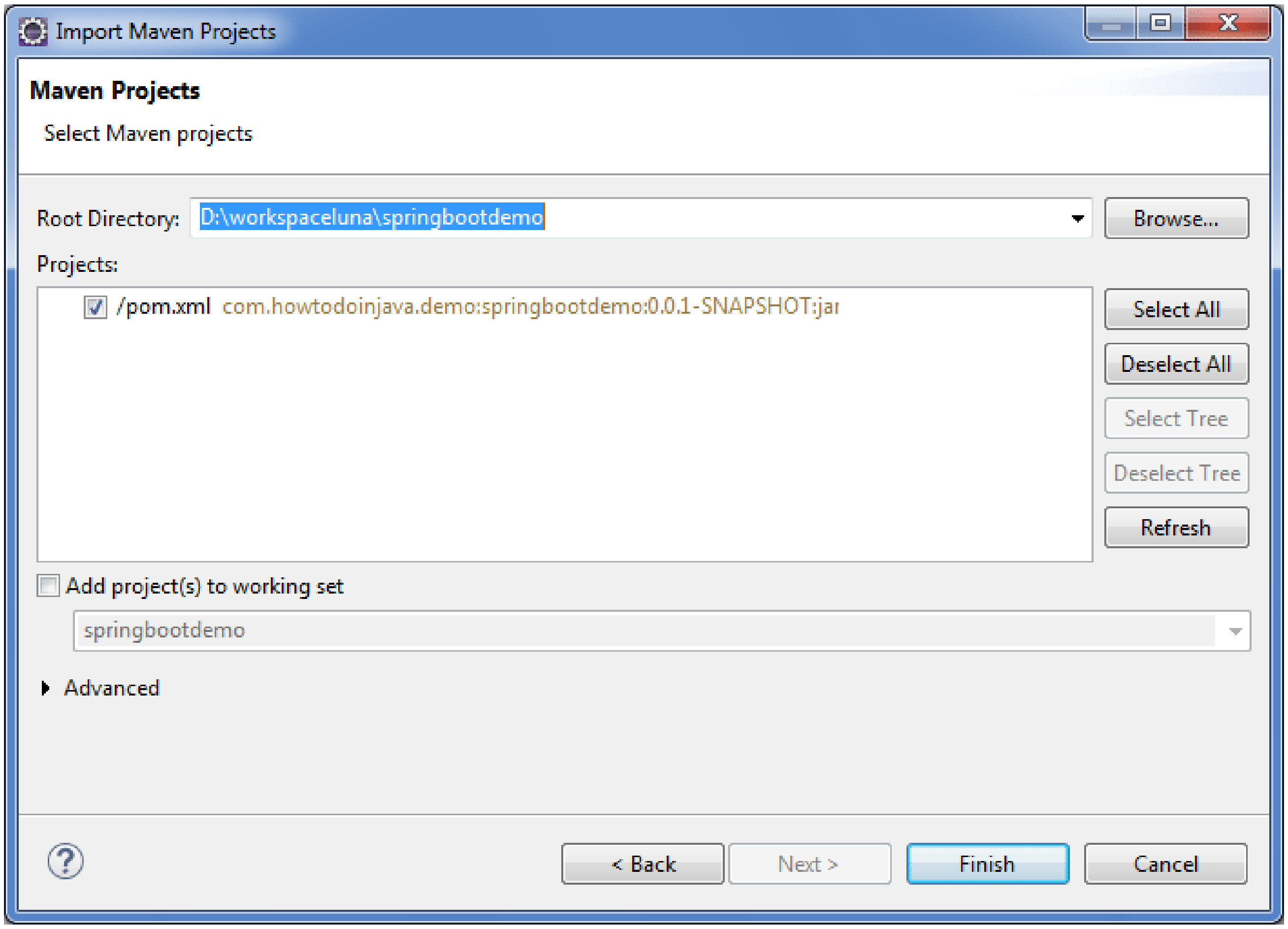
Next step is to import the generated project into your IDE. I have used eclipse for this purpose.

1) Import the spring boot project as an existing maven project.


Import
Select
Import Existing Maven Projects
Select an import source:
exi
General
Existing Projects into Workspace
Install
From Existing Installation
Maven
Existing Maven Projects
OpenShift
Existing OpenShift Application
< Back
Next >
Finish
Cancel

Import Existing Maven Project into Eclipse

2) Select the pom.xml file to import it.


Import Maven Projects

Maven Projects

Select Maven projects

Root Directory: D:\workspaceluna\springbootdemo

Projects:

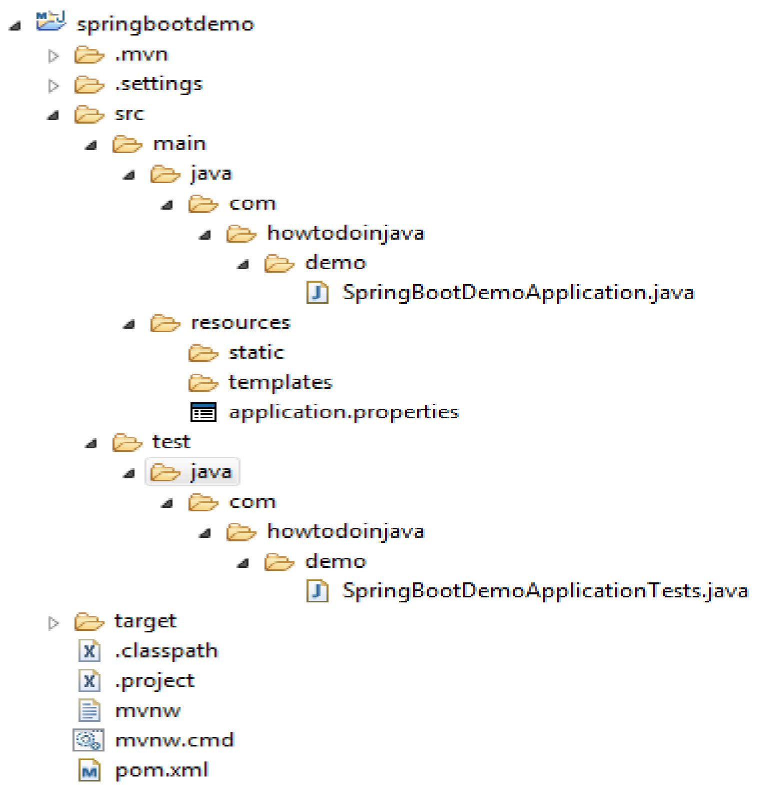
[✓] /pom.xml com.howtodoinjava.demo:springbootdemo:0.0.1-SNAPSHOT:jar

[ ] Add project(s) to working set

springbootdemo

Select pom.xml file to import maven project

3) Project will be imported and the dependencies you added while generating the zip file, will be automatically downloaded and added into classpath.



```
springbootdemo
  .mvn
  .settings
  src
    main
      java
        com
          howtodoinjava
            demo
              SpringBootDemoApplication.java
      resources
        static
        templates
        application.properties
    test
      java
        com
          howtodoinjava
            demo
              SpringBootDemoApplicationTests.java
  target
  .classpath
  .project
  mvnw
  mvnw.cmd
  pom.xml
```

Imported Spring Boot Project Structure

I have now successfully imported spring boot application.

Look into configuration.

3. Spring boot auto configuration

With spring boot, good thing is when you add a dependency (e.g. *Spring security*), it make fair assumptions and automatically configure some defaults for you. So you can start immediately.

Spring Boot uses convention over configuration by scanning the dependent libraries available in the class path. For each spring-boot-starter-* dependency in the POM file, Spring Boot executes a default AutoConfiguration class. AutoConfiguration classes use the *AutoConfiguration lexical pattern, where * represents the library. For example, the autoconfiguration of spring security is done through SecurityAutoConfiguration.
At the same time, if you don't want to use auto configuration for any project, it makes it very simple. Just use exclude = SecurityAutoConfiguration.class like below.

```
@SpringBootApplication (exclude = SecurityAutoConfiguration.class)

public class SpringBootDemoApplication {

  public static void main(String[] args)

  {

    SpringApplication.run(SpringBootDemoApplication.class, args);

  }

}
```

It is also possible to override default configuration values using the application.properties file in src/main/resources folder.

4. Spring boot annotations

Now look at @SpringBootApplication annotation what it actually does.

4.1. @SpringBootApplication annotation

SpringBootApplication is defined as below:

```
@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Inherited

@SpringBootConfiguration

@EnableAutoConfiguration

@ComponentScan(excludeFilters = @Filter(type = FilterType.CUSTOM, classes =
TypeExcludeFilter.class))

public @interface SpringBootApplication

{
   //more code
}
```

It adds 3 important annotations for application configuration purpose.

1. @SpringBootConfiguration

   ```
   @Configuration
   public @interface SpringBootConfiguration
   {
      //more code
   }
   ```

2. This annotation adds @Configuration annotation to class which **mark the class a source of bean definitions for the application context.**

3. @EnableAutoConfiguration

   This tells spring boot to auto configure important bean definitions based on added dependencies in pom.xml by start adding beans based on classpath settings, other beans, and various property settings.

4. @ComponentScan

   This annotation tells spring boot to scan the base package, find other beans/components and configure them as well.

5. verify auto-configured beans by spring boot

If you ever want to know what all beans have been automatically configured into your **spring boot hello world application**, then use this code and run it.

SpringBootDemoApplication.java

```
import java.util.Arrays;
```

```
import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.boot.autoconfigure.security.SecurityAutoConfiguration;

import org.springframework.context.ApplicationContext;


@SpringBootApplication (exclude = SecurityAutoConfiguration.class)

public class SpringBootDemoApplication {


  public static void main(String[] args)

  {

    ApplicationContext ctx = SpringApplication.run(SpringBootDemoApplication.class, args);


      String[] beanNames = ctx.getBeanDefinitionNames();


      Arrays.sort(beanNames);


      for (String beanName : beanNames)

      {

         System.out.println(beanName);

      }

  }
```

}

With my pom.xml file, it generates following beans names along with plenty of other springframework.boot.autoconfigure dependencies.

Console

```
simpleControllerHandlerAdapter

sortResolver

spring.datasource-org.springframework.boot.autoconfigure.jdbc.DataSourceProperties

spring.hateoas-org.springframework.boot.autoconfigure.hateoas.HateoasProperties

spring.http.encoding-org.springframework.boot.autoconfigure.web.HttpEncodingProperties

spring.http.multipart-org.springframework.boot.autoconfigure.web.MultipartProperties

spring.info-org.springframework.boot.autoconfigure.info.ProjectInfoProperties

spring.jackson-org.springframework.boot.autoconfigure.jackson.JacksonProperties

spring.jpa-org.springframework.boot.autoconfigure.orm.jpa.JpaProperties

spring.jta-org.springframework.boot.autoconfigure.transaction.jta.JtaProperties

spring.mvc-org.springframework.boot.autoconfigure.web.WebMvcProperties

spring.resources-org.springframework.boot.autoconfigure.web.ResourceProperties

springBootDemoApplication

standardJacksonObjectMapperBuilderCustomizer

stringHttpMessageConverter

tomcatEmbeddedServletContainerFactory

tomcatPoolDataSourceMetadataProvider

transactionAttributeSource

transactionInterceptor
```

transactionManager

transactionTemplate

viewControllerHandlerMapping

viewResolver

websocketContainerCustomizer

6. Using Spring boot REST API

Now it's time to build any functionality into application. We can add functionality as per the need, I am adding a REST API.

6.1. Create REST Controller

Create a package com.howtodoinjava.demo.controller and create rest controller inside it.

EmployeeController.java

```
import java.util.ArrayList;

import java.util.List;

import org.springframework.web.bind.annotation.RequestMapping;

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

import com.howtodoinjava.demo.model.Employee;


@RestController

public class EmployeeController

{

  @RequestMapping("/")

   public List<Employee> getEmployees()
```

```
    {
      List<Employee> employeesList = new ArrayList<Employee>();
      employeesList.add(new Employee(1,"lokesh","gupta","howtodoinjava@gmail.com"));
      return employeesList;
    }
}
```

6.2. Create Model

Create model class Employee.

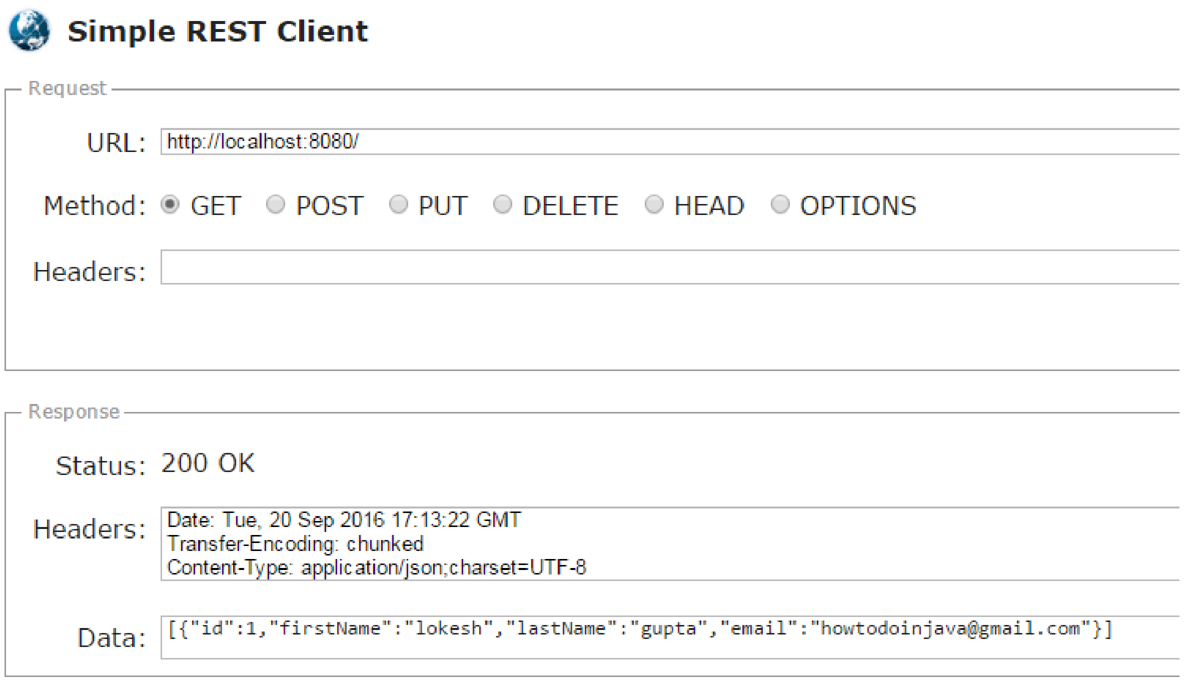
Employee.java

```
public class Employee {


  public Employee() {


  }
  public Employee(Integer id, String firstName, String lastName, String email) {
    super();
    this.id = id;
    this.firstName = firstName;
    this.lastName = lastName;
    this.email = email;
  }
```

```
   private Integer id;

   private String firstName;

   private String lastName;

   private String email;


   //getters and setters


   @Override

   public String toString() {

      return "Employee [id=" + id + ", firstName=" + firstName

            + ", lastName=" + lastName + ", email=" + email + "]";

   }

}
```

7. Spring boot hello world example demo

Now start the application by running main() method in SpringBootDemoApplication. It will start the embedded tomcat server on port 8080.
As we have configured the demo REST API URL to root URL, you can access it on http;//localhost:8080/ itself.



```
http://localhost:8080/
```

Status: 200 OK

```
Date: Tue, 20 Sep 2016 17:13:22 GMT
Transfer-Encoding: chunked
Content-Type: application/json;charset=UTF-8
```

```
[{"id":1,"firstName":"lokesh","lastName":"gupta","email":"howtodoinjava@gmail.com"}]
```

Verify Spring Boot REST API

You will get the below response in testing tool or browser.

```
[{"id":1,"firstName":"lokesh","lastName":"gupta","email":"howtodoinjava@gmail
.com"}]
```

That’s all for this **spring boot rest hello world example** with simple **rest api** example.

3.2. Add REST Endpoints

Add one RestController class which will expose a few endpoints like /elk, /elkdemo, /exception. Actually we are going to test a few log statements only, so feel free to add/modify logs as per your choice.

```
package com.example.howtodoinjava.elkexamplespringboot;

import java.io.PrintWriter;
import java.io.StringWriter;
import java.util.Date;
import org.apache.log4j.Level;
import org.apache.log4j.Logger;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.core.ParameterizedTypeReference;
import org.springframework.http.HttpMethod;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.client.RestTemplate;

@SpringBootApplication
public class ElkExampleSpringBootApplication {

    public static void main(String[] args) {
        SpringApplication.run(ElkExampleSpringBootApplication.class, args);
    }
}

@RestController
class ELKController {
    private static final Logger LOG = Logger.getLogger(ELKController.class.getName());

    @Autowired
    RestTemplate restTemplete;

    @Bean
    RestTemplate restTemplate() {
        return new RestTemplate();
    }

    @RequestMapping(value = "/elkdemo")
    public String helloWorld() {
        String response = "Hello user ! " + new Date();
        LOG.log(Level.INFO, "/elkdemo - &gt; " + response);

        return response;
    }
    @RequestMapping(value = "/elk")
    public String helloWorld1() {

        String response = restTemplete.exchange("http://localhost:8080/elkdemo", HttpMethod.GE
{
        }).getBody();
        LOG.log(Level.INFO, "/elk - &gt; " + response);

        try {
            String exceptionrsp = restTemplete.exchange("http://localhost:8080/exception", HttpMet
ParameterizedTypeReference() {
            }).getBody();
            LOG.log(Level.INFO, "/elk trying to print exception - &gt; " + exceptionrsp);
            response = response + " === " + exceptionrsp;
        } catch (Exception e) {
            // exception should not reach here. Really bad practice :)
        }

        return response;
    }

    @RequestMapping(value = "/exception")
    public String exception() {
        String rsp = "";
        try {
            int i = 1 / 0;
            // should get exception
        } catch (Exception e) {
            e.printStackTrace();
            LOG.error(e);

            StringWriter sw = new StringWriter();
            PrintWriter pw = new PrintWriter(sw);
            e.printStackTrace(pw);
            String sStackTrace = sw.toString(); // stack trace as a string
            LOG.error("Exception As String :: - &gt; "+sStackTrace);

            rsp = sStackTrace;
        }
        return rsp;
    }
}
```

3.3. Configure Spring boot Logging

Open application.properties under resources folder and add below configuration entries.

```
logging.file=elk-example.log
spring.application.name = elk-example
```

3.4. Verify Microservice Generated Logs

Do a final maven build using `mvn clean install` and start the application using command `java -jar target\elk-example-spring-boot-0.0.1-SNAPSHOT.jar` and test by browsing http://localhost:8080/elk.

Don't be afraid by seeing the big stack trace in the screen as it has been done intentionally to see how ELK handles exception message.

Go to the application root directory and verify that the log file i.e. elk-example.log has been created and do a couple of visits to the endpoints and verify that logs are getting added in the log file.

4. Logstash Configuration

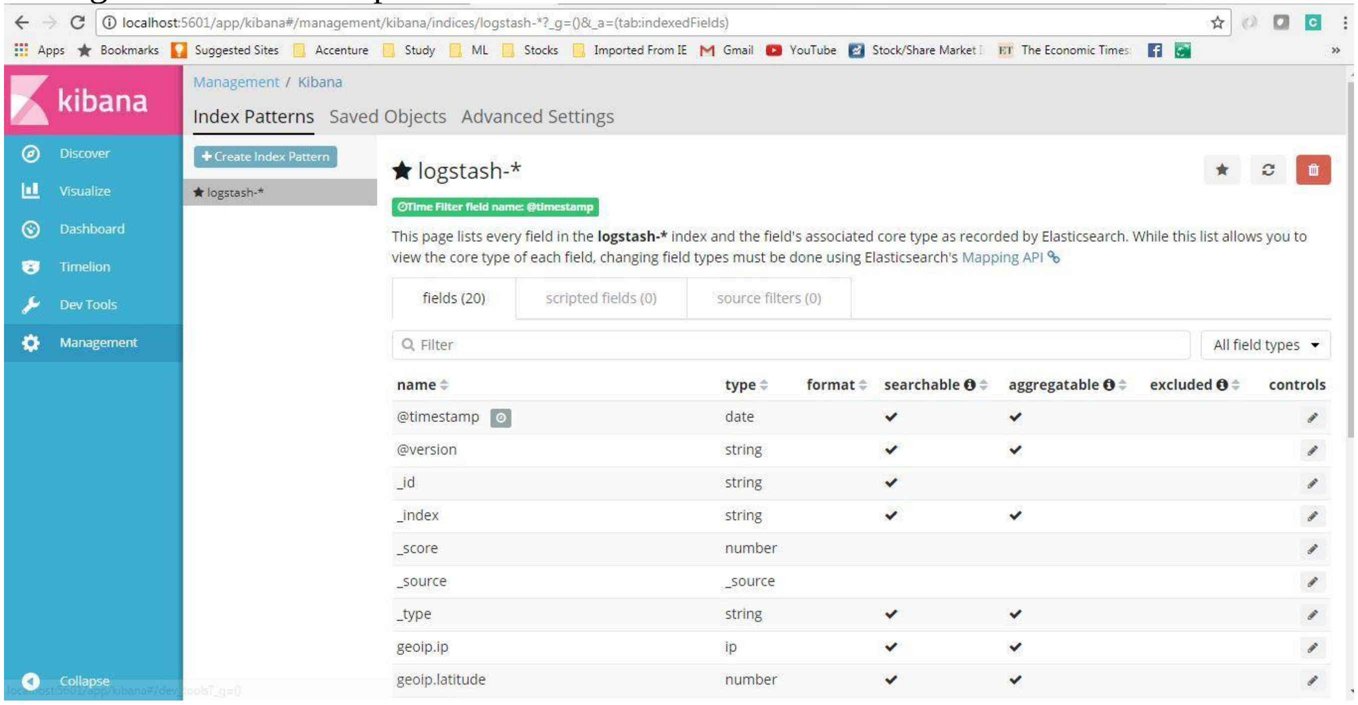
We need to create a logstash configuration file so that it listen to the log file and push log messages to elastic search. Here is the logstash configuration used in this project please change the log path as per your setup.

```
input {
  file {
    type => "java"
    path => "F:/Study/eclipse_workspace_mars/elk-example-spring-boot/elk-example.log"
    codec => multiline {
      pattern => "^%{YEAR}-%{MONTHNUM}-%{MONTHDAY} %{TIME}.*"
      negate => "true"
      what => "previous"
    }
  }
```

```
}

filter {
  #If log line contains tab character followed by 'at' then we will tag that entry as stacktrace
  if [message] =~ "\tat" {
    grok {
      match => ["message", "^(\tat)"]
      add_tag => ["stacktrace"]
    }
  }

 grok {
    match => [ "message",
               "(?<timestamp>%{YEAR}-%{MONTHNUM}-%{MONTHDAY} %{TIME})  %{L
9.]*\.(?<class>[A-Za-z0-9#_]+)\s*:\s+(?<logmessage>.*)",
               "message",
               "(?<timestamp>%{YEAR}-%{MONTHNUM}-%{MONTHDAY} %{TIME})  %{L
             ]
  }


  date {
    match => [ "timestamp" , "yyyy-MM-dd HH:mm:ss.SSS" ]
  }
}

output {

  stdout {
    codec => rubydebug
  }

  # Sending properly parsed log events to elasticsearch
  elasticsearch {
    hosts => ["localhost:9200"]
  }
}
```

5. Kibana Configuration

Before viewing the logs in Kibana, we need to configure the Index Patterns. We can configure logstash-* as default configuration. We can always change this index pattern in the logstash side and configure it in Kibana. For simplicity, we will work with default configuration.
The index pattern management page will look like below. With this configuration we are pointing Kibana to Elasticsearch index(s) of your choice. Logstash creates indices with the name pattern of logstash-YYYY.MM.DD We can do all those configuration in Kibana console http://localhost:5601/app/kibana and going to Management link in left panel.



★ logstash-*

Time Filter field name: @timestamp

This page lists every field in the **logstash-*** index and the field's associated core type as recorded by Elasticsearch. While this list allows you to view the core type of each field, changing field types must be done using Elasticsearch's Mapping API

fields (20) | scripted fields (0) | source filters (0)

name	type	format	searchable	aggregatable	excluded	controls
@timestamp	date		✓	✓		
@version	string		✓	✓		
_id	string		✓			
_index	string		✓	✓		
_score	number					
_source	_source					
_type	string		✓	✓		
geoip.ip	ip		✓	✓		
geoip.latitude	number		✓	✓		

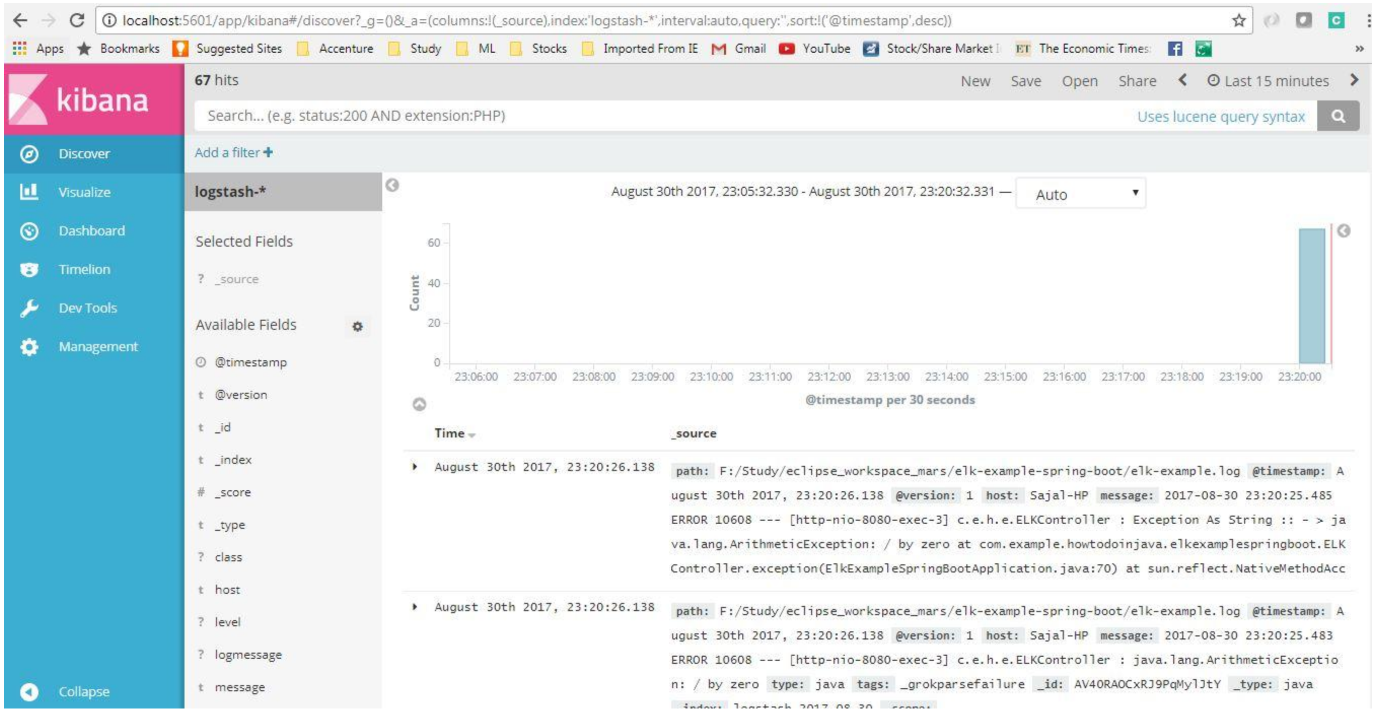
Logstash configuration in Kibana

6. Verify ELK Stack

Now when all components are up and running, let's verify the whole ecosystem.

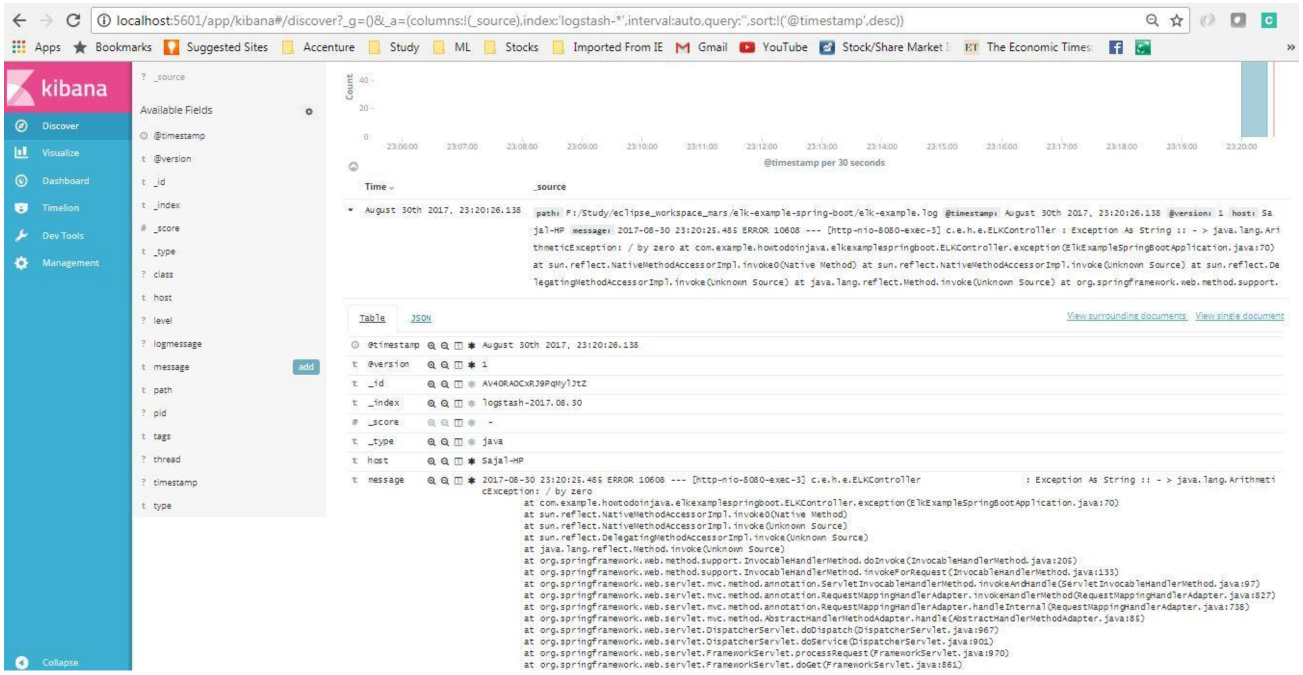
Go to application and test the end points couple of times so that logs got generated and then go to Kibana console and see that logs are properly stacked in the Kibana with lots of extra feature like we can filter, see different graphs etc in built.

Here is the view of generated logs in Kibana.


localhost:5601/app/kibana#/discover?_g=()&_a=(columns:!(_source),index:'logstash-*',interval:auto,query:'',sort:!('@timestamp',desc))
kibana
Discover
Visualize
Dashboard
Timelion
Dev Tools
Management
Collapse
67 hits
New
Save
Open
Share
Last 15 minutes
Search... (e.g. status:200 AND extension:PHP)
Uses lucene query syntax
Add a filter
logstash-*
Selected Fields
_source
Available Fields
@timestamp
@version
_id
_index
_score
_type
class
host
level
logmessage
message
August 30th 2017, 23:05:32.330 - August 30th 2017, 23:20:32.331 —
Auto
Count
@timestamp per 30 seconds

```
Time                              _source
August 30th 2017, 23:20:26.138    path: F:/Study/eclipse_workspace_mars/elk-example-spring-boot/elk-example.log @timestamp: A
                                  ugust 30th 2017, 23:20:26.138 @version: 1 host: Sajal-HP message: 2017-08-30 23:20:25.485
                                  ERROR 10608 --- [http-nio-8080-exec-3] c.e.h.e.ELKController : Exception As String :: - > ja
                                  va.lang.ArithmeticException: / by zero at com.example.howtodoinjava.elkexamplespringboot.ELK
                                  Controller.exception(ElkExampleSpringBootApplication.java:70) at sun.reflect.NativeMethodAcc
August 30th 2017, 23:20:26.138    path: F:/Study/eclipse_workspace_mars/elk-example-spring-boot/elk-example.log @timestamp: A
                                  ugust 30th 2017, 23:20:26.138 @version: 1 host: Sajal-HP message: 2017-08-30 23:20:25.483
                                  ERROR 10608 --- [http-nio-8080-exec-3] c.e.h.e.ELKController : java.lang.ArithmeticExceptio
                                  n: / by zero type: java tags: _grokparsefailure _id: AV40RAOCxRJ9PqMylJtY _type: java
```

Kibana Logs Overview



```
Time                              _source
August 30th 2017, 23:20:26.138    path: F:/Study/eclipse_workspace_mars/elk-example-spring-boot/elk-example.log @timestamp: August 30th 2017, 23:20:26.138 @version: 1 host: Sa
                                  jal-HP message: 2017-08-30 23:20:25.485 ERROR 10608 --- [http-nio-8080-exec-3] c.e.h.e.ELKController : Exception As String :: - > java.lang.Ari
                                  thmeticException: / by zero at com.example.howtodoinjava.elkexamplespringboot.ELKController.exception(ElkExampleSpringBootApplication.java:70)
                                  at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method) at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source) at sun.reflect.De
                                  legatingMethodAccessorImpl.invoke(Unknown Source) at java.lang.reflect.Method.invoke(Unknown Source) at org.springframework.web.method.support.

@timestamp   August 30th 2017, 23:20:26.138
@version     1
_id          AV40RAOCxRJ9PqMylJtZ
_index       logstash-2017.08.30
_score       -
_type        java
host         Sajal-HP
message      2017-08-30 23:20:25.485 ERROR 10608 --- [http-nio-8080-exec-3] c.e.h.e.ELKController                 : Exception As String :: - > java.lang.Arithmeti
             cException: / by zero
                     at com.example.howtodoinjava.elkexamplespringboot.ELKController.exception(ElkExampleSpringBootApplication.java:70)
                     at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
                     at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
                     at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
                     at java.lang.reflect.Method.invoke(Unknown Source)
                     at org.springframework.web.method.support.InvocableHandlerMethod.doInvoke(InvocableHandlerMethod.java:205)
                     at org.springframework.web.method.support.InvocableHandlerMethod.invokeForRequest(InvocableHandlerMethod.java:133)
                     at org.springframework.web.servlet.mvc.method.annotation.ServletInvocableHandlerMethod.invokeAndHandle(ServletInvocableHandlerMethod.java:97)
                     at org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter.invokeHandlerMethod(RequestMappingHandlerAdapter.java:827)
                     at org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter.handleInternal(RequestMappingHandlerAdapter.java:738)
                     at org.springframework.web.servlet.mvc.method.AbstractHandlerMethodAdapter.handle(AbstractHandlerMethodAdapter.java:85)
                     at org.springframework.web.servlet.DispatcherServlet.doDispatch(DispatcherServlet.java:967)
                     at org.springframework.web.servlet.DispatcherServlet.doService(DispatcherServlet.java:901)
                     at org.springframework.web.servlet.FrameworkServlet.processRequest(FrameworkServlet.java:970)
                     at org.springframework.web.servlet.FrameworkServlet.doGet(FrameworkServlet.java:861)
```

Kibana Logs details screen

IV Automate Docker build and deployment using Jenkins pipeline code

On Windows

The Jenkins project provides a Linux container image, not a Windows container image. Be sure that your Docker for Windows installation is configured to run Linux Containers rather than Windows Containers.. Once configured to run Linux Containers, the steps are:

1. Open up a command prompt window do the following:
2. Create a bridge network in Docker
   docker network create jenkins
3. Run a docker:dind Docker image
4. docker run --name jenkins-docker --rm --detach ^
5.   --privileged --network jenkins --network-alias docker ^
6.   --env DOCKER_TLS_CERTDIR=/certs ^
7.   --volume jenkins-docker-certs:/certs/client ^
8.   --volume jenkins-data:/var/jenkins_home ^
     docker:dind
9. Build a customised official Jenkins Docker image using above Dockerfile and docker build command.
10. Run your own myjenkins-blueocean:1.1 image as a container in Docker using the following docker run command:
11. docker run --name jenkins-blueocean --rm --detach ^
12.   --network jenkins --env DOCKER_HOST=tcp://docker:2376 ^
13.   --env DOCKER_CERT_PATH=/certs/client --env DOCKER_TLS_VERIFY=1 ^
14.   --volume jenkins-data:/var/jenkins_home ^
15.   --volume jenkins-docker-certs:/certs/client:ro ^
16.   --volume "%HOMEDRIVE%%HOMEPATH%":/home ^
      --publish 8080:8080 --publish 50000:50000 myjenkins-blueocean:1.1
17. Proceed to the Setup wizard.

Accessing the Docker container

To access Docker container through a terminal/command prompt using the docker exec command, you can add an option like --name jenkins-tutorial to the docker exec command. That will access the Jenkins Docker container named "jenkins-tutorial".

This means you could access your docker container (through a separate terminal/command prompt window) with a docker exec command like:

```
docker exec -it jenkins-blueocean bash
```

Accessing the Docker logs

There is a possibility you may need to access the Jenkins console log, for instance, when Unlocking Jenkins as part of the Post-installation setup wizard.

The Jenkins console log is easily accessible through the terminal/command prompt window from which you executed the docker run … command. In case if needed you can also access the Jenkins console log through the Docker logs of your container using the following command:

```
docker logs <docker-container-name>
```

Your <docker-container-name> can be obtained using the docker ps command.

Accessing the Jenkins home directory

There is a possibility you may need to access the Jenkins home directory, for instance, to check the details of a Jenkins build in the workspace subdirectory.

If you mapped the Jenkins home directory (/var/jenkins_home) to one on your machine's local file system (i.e. in the docker run … command above), then you can access the contents of this directory through your machine's usual terminal/command prompt.

Otherwise, if you specified the --volume jenkins-data:/var/jenkins_home option in the docker run … command, you can access the contents of the Jenkins home directory through your container's terminal/command prompt using the docker container exec command:

```
docker container exec -it <docker-container-name> bash
```

As mentioned above, your <docker-container-name> can be obtained using the docker container ls command. If you specified the

--name jenkins-blueocean option in the docker container run …command above (see also Accessing the Jenkins/Blue Ocean Docker container), you can simply use the docker container exec command:

```
docker container exec -it jenkins-blueocean bash
```

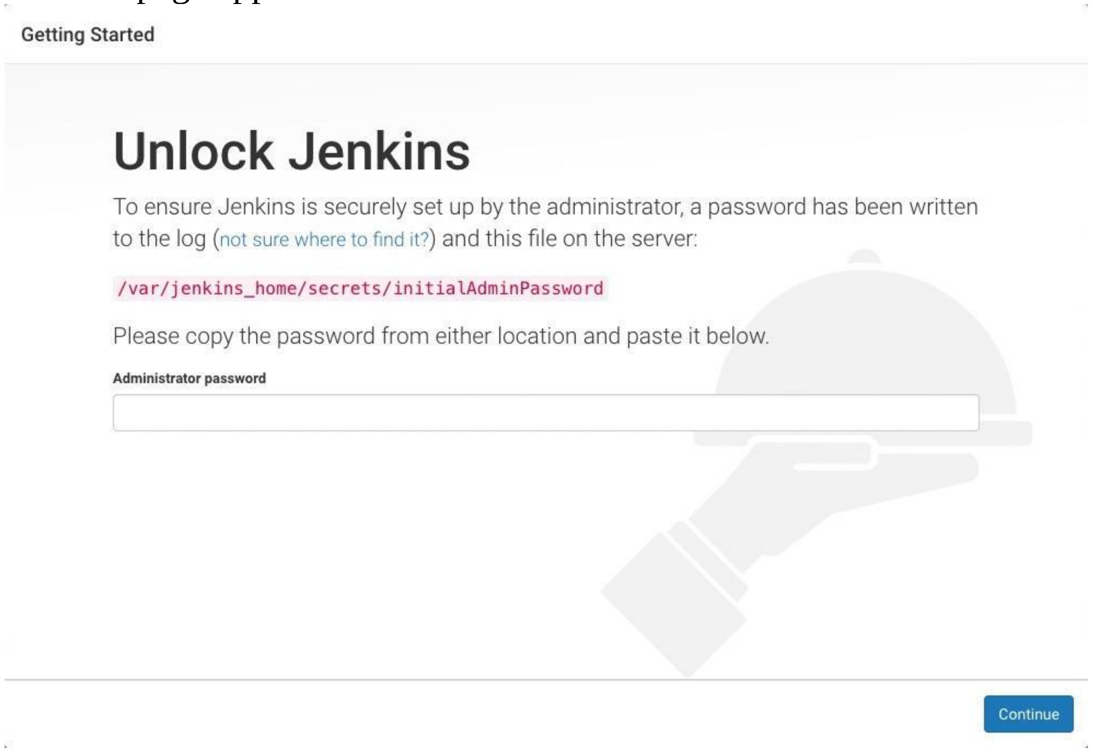
Setup wizard

Before you can access Jenkins, there are a few quick "one-off" steps you'll need to perform.

Unlocking Jenkins

When you first access a new Jenkins instance, you are asked to unlock it using an automatically-generated password.

1. After the 2 sets of asterisks appear in the terminal/command prompt window, browse to http://localhost:8080 and wait until the **Unlock Jenkins** page appears.



Getting Started

Unlock Jenkins

To ensure Jenkins is securely set up by the administrator, a password has been written to the log (not sure where to find it?) and this file on the server:

`/var/jenkins_home/secrets/initialAdminPassword`

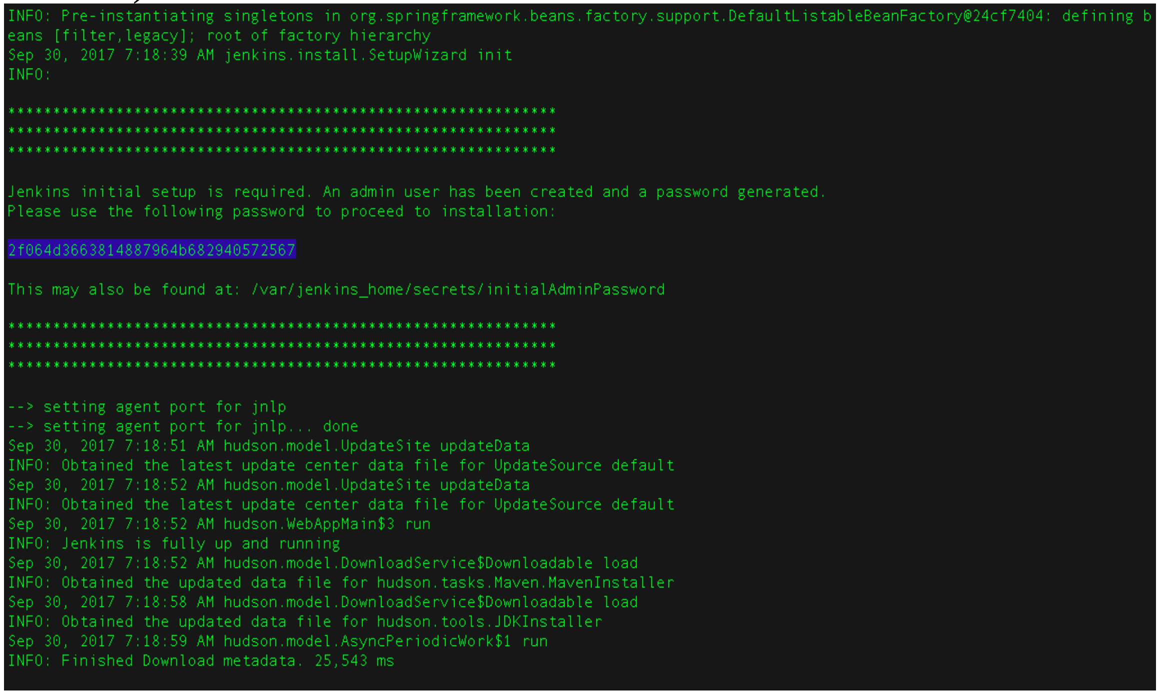
Please copy the password from either location and paste it below.

Administrator password

2. Display the Jenkins console log with the command:

```
docker logs jenkins-blueocean
```

3. From your terminal/command prompt window again, copy the automatically-generated alphanumeric password (between the 2 sets of asterisks).



```
INFO: Pre-instantiating singletons in org.springframework.beans.factory.support.DefaultListableBeanFactory@24cf7404: defining b
eans [filter,legacy]; root of factory hierarchy
Sep 30, 2017 7:18:39 AM jenkins.install.SetupWizard init
INFO:

*************************************************************
*************************************************************
*************************************************************

Jenkins initial setup is required. An admin user has been created and a password generated.
Please use the following password to proceed to installation:

2f064d3663814887964b682940572567

This may also be found at: /var/jenkins_home/secrets/initialAdminPassword

*************************************************************
*************************************************************
*************************************************************

--> setting agent port for jnlp
--> setting agent port for jnlp... done
Sep 30, 2017 7:18:51 AM hudson.model.UpdateSite updateData
INFO: Obtained the latest update center data file for UpdateSource default
Sep 30, 2017 7:18:52 AM hudson.model.UpdateSite updateData
INFO: Obtained the latest update center data file for UpdateSource default
Sep 30, 2017 7:18:52 AM hudson.WebAppMain$3 run
INFO: Jenkins is fully up and running
Sep 30, 2017 7:18:52 AM hudson.model.DownloadService$Downloadable load
INFO: Obtained the updated data file for hudson.tasks.Maven.MavenInstaller
Sep 30, 2017 7:18:58 AM hudson.model.DownloadService$Downloadable load
INFO: Obtained the updated data file for hudson.tools.JDKInstaller
Sep 30, 2017 7:18:59 AM hudson.model.AsyncPeriodicWork$1 run
INFO: Finished Download metadata. 25,543 ms
```

4. On the **Unlock Jenkins** page, paste this password into the **Administrator password** field and click **Continue**.

Customizing Jenkins with plugins

After unlocking Jenkins, the **Customize Jenkins** page appears.

On this page, click **Install suggested plugins**.

The setup wizard shows the progression of Jenkins being configured and the suggested plugins being installed. This process may take a few minutes.

Creating the first administrator user

Finally, Jenkins asks you to create your first administrator user.

1. When the **Create First Admin User** page appears, specify your details in the respective fields and click **Save and Finish**.
2. When the **Jenkins is ready** page appears, click **Start using Jenkins**.
   Notes:
   - This page may indicate **Jenkins is almost ready!** instead and if so, click **Restart**.

    - If the page doesn't automatically refresh after a minute, use your web browser to refresh the page manually.
3. If required, log in to Jenkins with the credentials of the user you just created and you're ready to start using Jenkins!

Stopping and restarting Jenkins

Throughout the remainder of this tutorial, you can stop your Docker container by running:

```
docker stop jenkins-blueocean jenkins-docker
```

To restart your Docker container:

1. Run the same docker run … commands you ran for macOS, Linux or Windows above.
2. Browse to http://localhost:8080.
3. Wait until the log in page appears and log in.

Fork and clone the sample repository

Obtain the Java application from GitHub, by forking the sample repository of the application's source code into your own GitHub account and then cloning this fork locally.

1. Ensure you are signed in to your GitHub account. If you don't yet have a GitHub account, sign up for a free one on the GitHub website.
2. Fork the simple-java-maven-app on GitHub into your local GitHub account. If you need help with this process, refer to the Fork A Repo documentation on the GitHub website for more information.
3. Clone your forked simple-java-maven-app repository (on GitHub) locally to your machine. To begin this process, do either of the following (where <your-username> is the name of your user account on your operating system):
    - If you have the GitHub Desktop app installed on your machine:
        a. In GitHub, click the green **Clone or download** button on your forked repository, then **Open in Desktop**.
        b. In GitHub Desktop, before clicking **Clone** on the **Clone a Repository** dialog box, ensure **Local Path** for:
            - macOS is /Users/<your-username>/Documents/GitHub/simple-java-maven-app
            - Linux is /home/<your-username>/GitHub/simple-java-maven-app

        - Windows is C:\Users\<your-username>\Documents\GitHub\simple-java-maven-app
  - Otherwise:
    a. Open up a terminal/command line prompt and cd to the appropriate directory on:
        - macOS - /Users/<your-username>/Documents/GitHub/
        - Linux - /home/<your-username>/GitHub/
        - Windows - C:\Users\<your-username>\Documents\GitHub\ (although use a Git bash command line window as opposed to the usual Microsoft command prompt)

    b. Run the following command to continue/complete cloning your forked repo:
       git clone https://github.com/YOUR-GITHUB-ACCOUNT-NAME/simple-java-maven-app
       where YOUR-GITHUB-ACCOUNT-NAME is the name of your GitHub account.

Create your Pipeline project in Jenkins

1. Go back to Jenkins, log in again if necessary and click **create new jobs** under **Welcome to Jenkins!**
   Note: If you don't see this, click **New Item** at the top left.
2. In the **Enter an item name** field, specify the name for your new Pipeline project (e.g. simple-java-maven-app).
3. Scroll down and click **Pipeline**, then click **OK** at the end of the page.
4. (*Optional*) On the next page, specify a brief description for your Pipeline in the **Description** field (e.g. An entry-level Pipeline demonstrating how to use Jenkins to build a simple Java application with Maven.)
5. Click the **Pipeline** tab at the top of the page to scroll down to the **Pipeline** section.
6. From the **Definition** field, choose the **Pipeline script from SCM** option. This option instructs Jenkins to obtain your Pipeline from Source Control Management (SCM), which will be your locally cloned Git repository.
7. From the **SCM** field, choose **Git**.
8. In the **Repository URL** field, specify the directory path of your locally cloned repository above, which is from your user account/home directory on your host machine, mapped to the /home directory of the Jenkins container - i.e.
   - For Windows - /home/Documents/GitHub/simple-java-maven-app

9. Click **Save** to save your new Pipeline project. You’re now ready to begin creating your Jenkinsfile, which you’ll be checking into your locally cloned Git repository.

Create your initial Pipeline as a Jenkinsfile

You’re now ready to create your Pipeline that will automate building your Java application with Maven in Jenkins. Your Pipeline will be created as a Jenkinsfile, which will be committed to your locally cloned Git repository (simple-java-maven-app).

This is the foundation of "Pipeline-as-Code", which treats the continuous delivery pipeline as a part of the application to be versioned and reviewed like any other code. Read more about Pipeline and what a Jenkinsfile is in the Pipeline and Using a Jenkinsfile sections of the User Handbook.

First, create an initial Pipeline to download a Maven Docker image and run it as a Docker container (which will build your simple Java application). Also add a "Build" stage to the Pipeline that begins orchestrating this whole process.

1. Using your favorite text editor or IDE, create and save new text file with the name Jenkinsfile at the root of your local simple-java-maven-app Git repository.
2. Copy the following Declarative Pipeline code and paste it into your empty Jenkinsfile:

```
pipeline {
    agent {
        docker {
            image 'maven:3-alpine'
            args '-v /root/.m2:/root/.m2'
        }
    }
    stages {
        stage('Build') {
            steps {
                sh 'mvn -B -DskipTests clean package'
            }
        }
    }
}
```

This image parameter (of the agent section's docker parameter) downloads the maven:3-alpine Docker image (if it's not already available on your machine) and runs this image as a separate container. This means that:

- You'll have separate Jenkins and Maven containers running locally in Docker.
- The Maven container becomes the agent that Jenkins uses to run your Pipeline project. However, this container is short-lived - its lifespan is only that of the duration of your Pipeline's execution.

This args parameter creates a reciprocal mapping between the /root/.m2 (i.e. Maven repository) directories in the short-lived Maven Docker container and that of your Docker host's filesystem. Explaining the details behind this is beyond the scope of this tutorial. However, the main reason for doing this is to ensure that the artifacts necessary to build your Java application (which Maven downloads while your Pipeline is being executed) are retained in the Maven repository beyond the lifespan of the Maven container. This prevents Maven from having to download the same artifacts during successive runs of your Jenkins Pipeline, which you'll be conducting later on. Be aware that unlike the Docker data volume you created for jenkins-data above, the Docker host's filesystem is effectively cleared out each time Docker is restarted. This means you'll lose the downloaded Maven repository artifacts each time Docker restarts.

Defines a stage (directive) called Build that appears on the Jenkins UI.

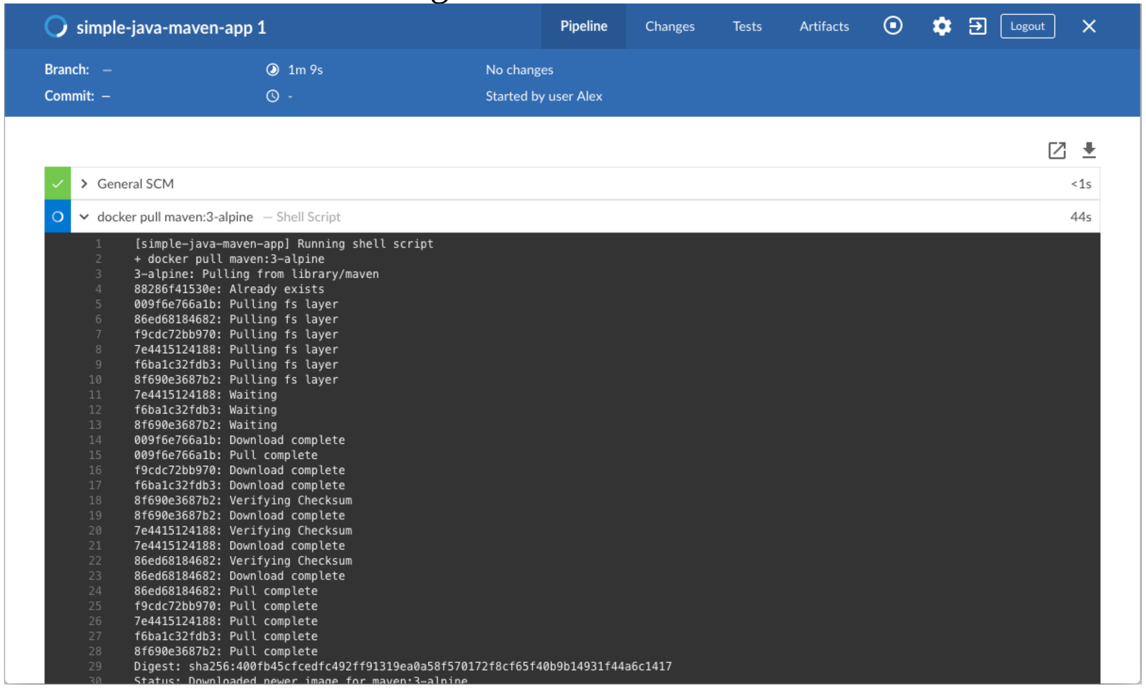
This sh step (of the steps section) runs the Maven command to cleanly build your Java application (without running any tests).

17. Save your edited Jenkinsfile and commit it to your local simple-java-maven-app Git repository. E.g. Within the simple-java-maven-app directory, run the commands:
    git add .
    then
    git commit -m "Add initial Jenkinsfile"
18. Go back to Jenkins again, log in again if necessary and click **Open Blue Ocean** on the left to access Jenkins's Blue Ocean interface.

19. In the **This job has not been run** message box, click **Run**, then quickly click the **OPEN** link which appears briefly at the lower-right to see Jenkins running your Pipeline project. If you weren't able to click the **OPEN** link, click the row on the main Blue Ocean interface to access this feature. **Note:** You may need to wait several minutes for this first run to complete. After making a clone of your local simple-java-maven-app Git repository itself, Jenkins:

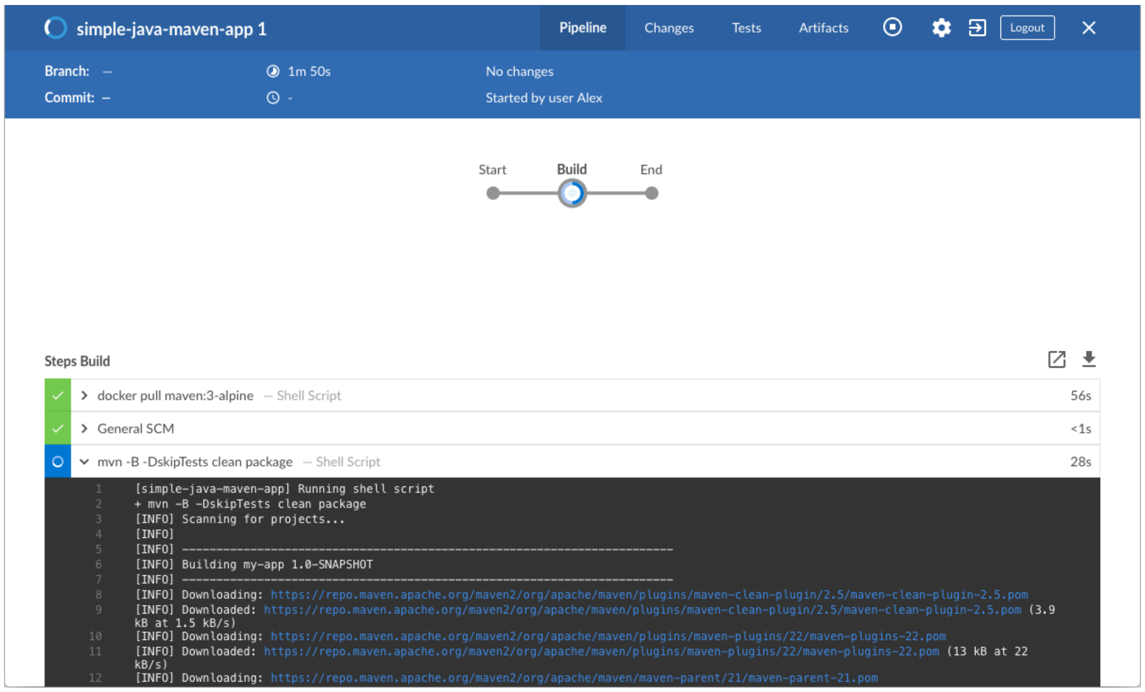
a. Initially queues the project to be run on the agent.

b. Downloads the Maven Docker image and runs it in a container on Docker.



```
[simple-java-maven-app] Running shell script
+ docker pull maven:3-alpine
3-alpine: Pulling from library/maven
88286f41530e: Already exists
009f6e766a1b: Pulling fs layer
86ed68184682: Pulling fs layer
f9cdc72bb970: Pulling fs layer
7e4415124188: Pulling fs layer
f6ba1c32fdb3: Pulling fs layer
8f690e3687b2: Pulling fs layer
7e4415124188: Waiting
f6ba1c32fdb3: Waiting
8f690e3687b2: Waiting
009f6e766a1b: Download complete
009f6e766a1b: Pull complete
f9cdc72bb970: Download complete
f6ba1c32fdb3: Download complete
8f690e3687b2: Verifying Checksum
8f690e3687b2: Download complete
7e4415124188: Verifying Checksum
7e4415124188: Download complete
86ed68184682: Verifying Checksum
86ed68184682: Download complete
86ed68184682: Pull complete
f9cdc72bb970: Pull complete
7e4415124188: Pull complete
f6ba1c32fdb3: Pull complete
8f690e3687b2: Pull complete
Digest: sha256:400fb45cfcedfc492ff91319ea0a58f570172f8cf65f40b9b14931f44a6c1417
Status: Downloaded newer image for maven:3-alpine
```

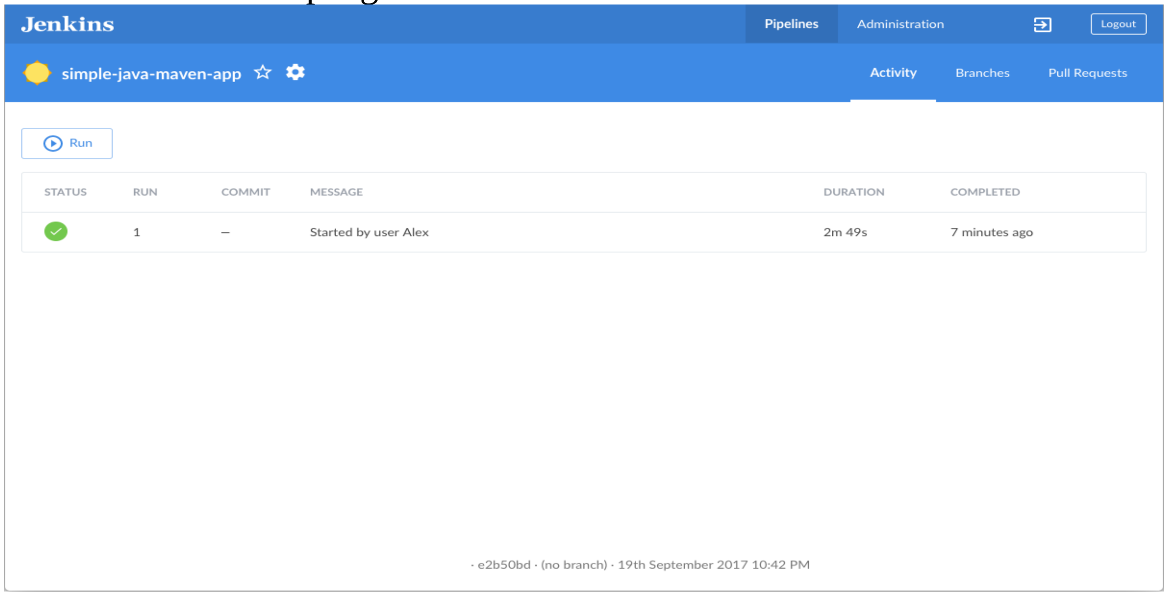
c. Runs the Build stage (defined in the Jenkinsfile) on the Maven container. During this time, Maven downloads many artifacts necessary to build your Java application, which will ultimately be stored in Jenkins's local Maven repository (in the Docker host's filesystem).



```
[simple-java-maven-app] Running shell script
+ mvn -B -DskipTests clean package
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building my-app 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO] Downloading: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-clean-plugin/2.5/maven-clean-plugin-2.5.pom
[INFO] Downloaded: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-clean-plugin/2.5/maven-clean-plugin-2.5.pom (3.9
kB at 1.5 kB/s)
[INFO] Downloading: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-plugins/22/maven-plugins-22.pom
[INFO] Downloaded: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-plugins/22/maven-plugins-22.pom (13 kB at 22
kB/s)
[INFO] Downloading: https://repo.maven.apache.org/maven2/org/apache/maven/maven-parent/21/maven-parent-21.pom
```

2. The Blue Ocean interface turns green if Jenkins built your Java application successfully.

3. Click the **X** at the top-right to return to the main Blue Ocean interface.



STATUS	RUN	COMMIT	MESSAGE	DURATION	COMPLETED
✓	1	–	Started by user Alex	2m 49s	7 minutes ago

· e2b50bd · (no branch) · 19th September 2017 10:42 PM

Add a test stage to your Pipeline

1. Go back to your text editor/IDE and ensure your Jenkinsfile is open.
2. Copy and paste the following Declarative Pipeline syntax immediately under the Build stage of your Jenkinsfile:

```
        stage('Test') {
            steps {
                sh 'mvn test'
            }
            post {
                always {
                    junit 'target/surefire-reports/*.xml'
                }
            }
        }
```

so that you end up with:

```
pipeline {
    agent {
        docker {
            image 'maven:3-alpine'
            args '-v /root/.m2:/root/.m2'
        }
    }
    stages {
        stage('Build') {
            steps {
                sh 'mvn -B -DskipTests clean package'
            }
        }
        stage('Test') {
            steps {
                sh 'mvn test'
            }
            post {
                always {
                    junit 'target/surefire-reports/*.xml'
                }
            }
        }
    }
}
```

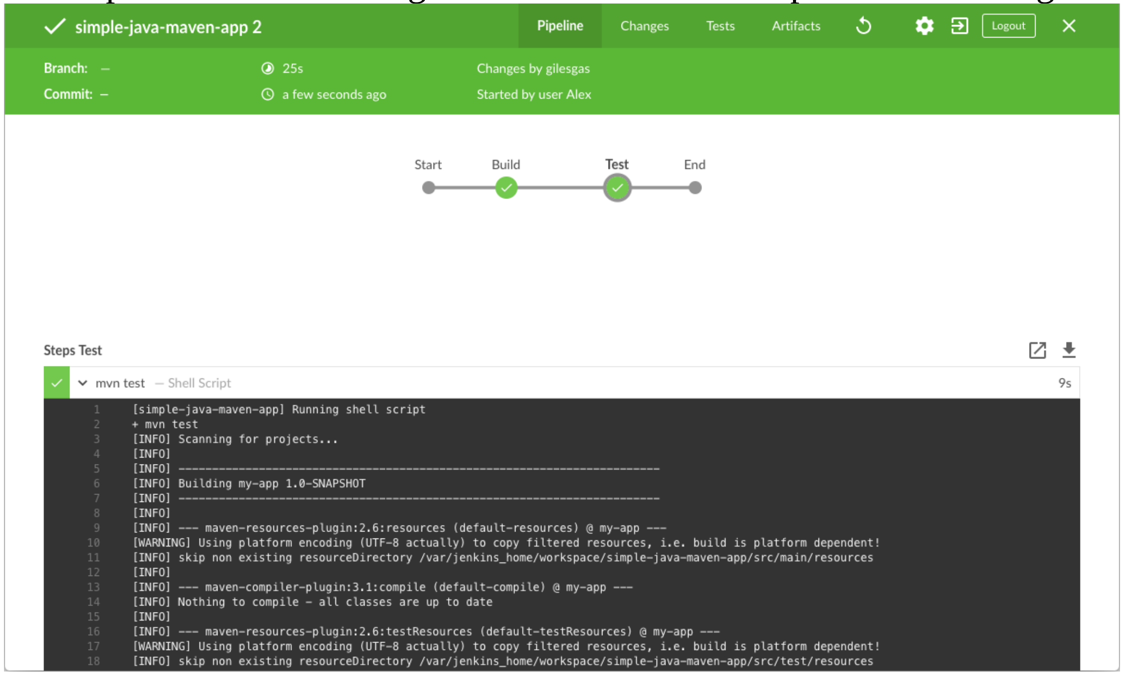
Defines a stage (directive) called Test that appears on the Jenkins UI.

This sh step (of the steps section) executes the Maven command to run the unit test on your simple Java application. This command also generates a JUnit XML report, which is saved to the target/surefire-reports directory (within the /var/jenkins_home/workspace/simple-java-maven-app directory in the Jenkins container).

This junit step (provided by the JUnit Plugin) archives the JUnit XML report (generated by the mvn test command above) and exposes the results through the Jenkins interface. In Blue Ocean, the results are accessible through the **Tests** page of a Pipeline run.
The post section's always condition that contains this junit step ensures that the step is *always* executed *at the completion* of the Test stage, regardless of the stage's outcome.

12. Save your edited Jenkinsfile and commit it to your local simple-java-maven-app Git repository. E.g. Within the simple-java-maven-app directory, run the commands:
    git stage .
    then
    git commit -m "Add 'Test' stage"
13. Go back to Jenkins again, log in again if necessary and ensure you've accessed Jenkins's Blue Ocean interface.
14. Click **Run** at the top left, then quickly click the **OPEN** link which appears briefly at the lower-right to see Jenkins running your amended Pipeline project. If you weren't able to click the **OPEN** link, click the *top* row on the Blue Ocean interface to access this feature.
    Note: You'll notice from this run that Jenkins no longer needs to download the Maven Docker image. Instead, Jenkins only needs to run a new container from the Maven image downloaded previously. Also, if Docker had not restarted since you last ran the Pipeline above, then no Maven artifacts need to be downloaded during the "Build" stage. Therefore, running your Pipeline this subsequent time should be much faster.

If your amended Pipeline ran successfully, here's what the Blue Ocean interface should look like. Notice the additional "Test" stage. You can click on the previous "Build" stage circle to access the output from that stage.



```
[simple-java-maven-app] Running shell script
+ mvn test
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building my-app 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ my-app ---
[WARNING] Using platform encoding (UTF-8 actually) to copy filtered resources, i.e. build is platform dependent!
[INFO] skip non existing resourceDirectory /var/jenkins_home/workspace/simple-java-maven-app/src/main/resources
[INFO]
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ my-app ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-resources-plugin:2.6:testResources (default-testResources) @ my-app ---
[WARNING] Using platform encoding (UTF-8 actually) to copy filtered resources, i.e. build is platform dependent!
[INFO] skip non existing resourceDirectory /var/jenkins_home/workspace/simple-java-maven-app/src/test/resources
```

15. Click the **X** at the top-right to return to the main Blue Ocean interface.

Add a final deliver stage to your Pipeline

1. Go back to your text editor/IDE and ensure your Jenkinsfile is open.
2. Copy and paste the following Declarative Pipeline syntax immediately under the Test stage of your Jenkinsfile:

```
3.        stage('Deliver') {
4.            steps {
5.                sh './jenkins/scripts/deliver.sh'
6.            }
          }
```

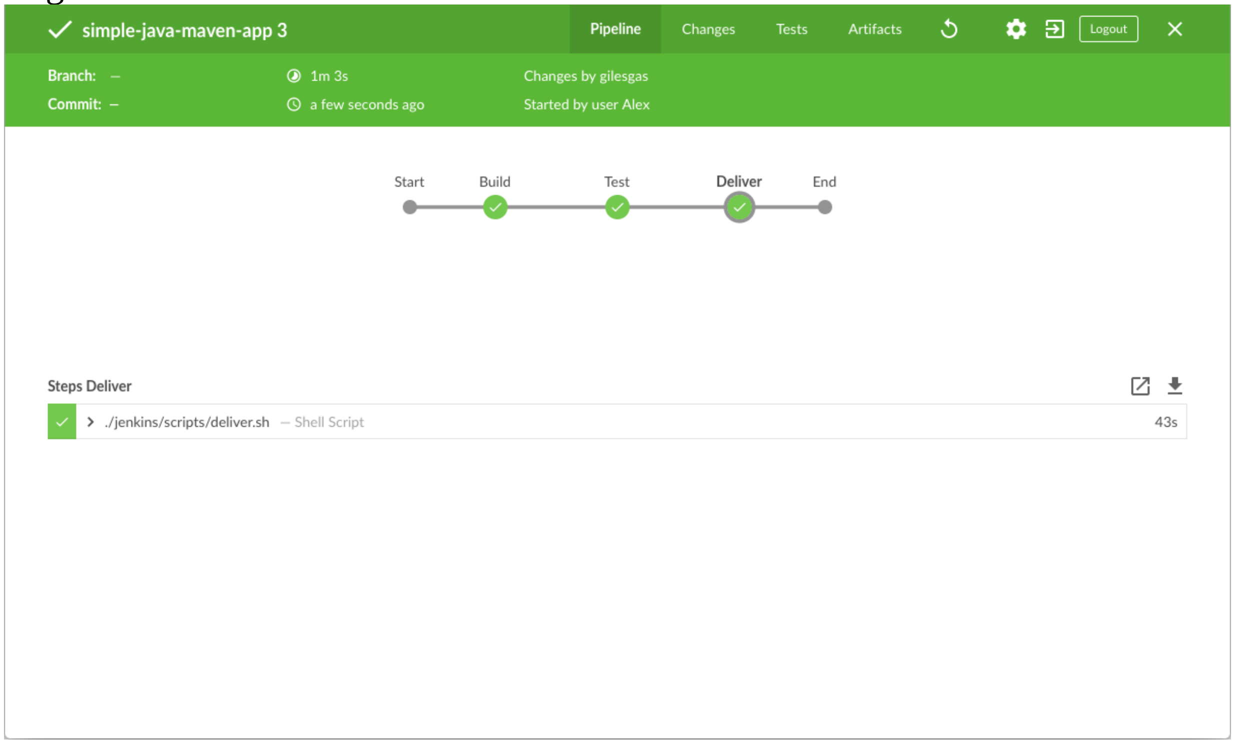
and add a skipStagesAfterUnstable option so that you end up with:

```
pipeline {
    agent {
        docker {
            image 'maven:3-alpine'
            args '-v /root/.m2:/root/.m2'
        }
    }
    options {
        skipStagesAfterUnstable()
    }
    stages {
        stage('Build') {
            steps {
                sh 'mvn -B -DskipTests clean package'
            }
        }
        stage('Test') {
            steps {
                sh 'mvn test'
            }
            post {
                always {
                    junit 'target/surefire-reports/*.xml'
                }
            }
        }
        stage('Deliver') {
            steps {
                sh './jenkins/scripts/deliver.sh'
            }
        }
    }
}
```

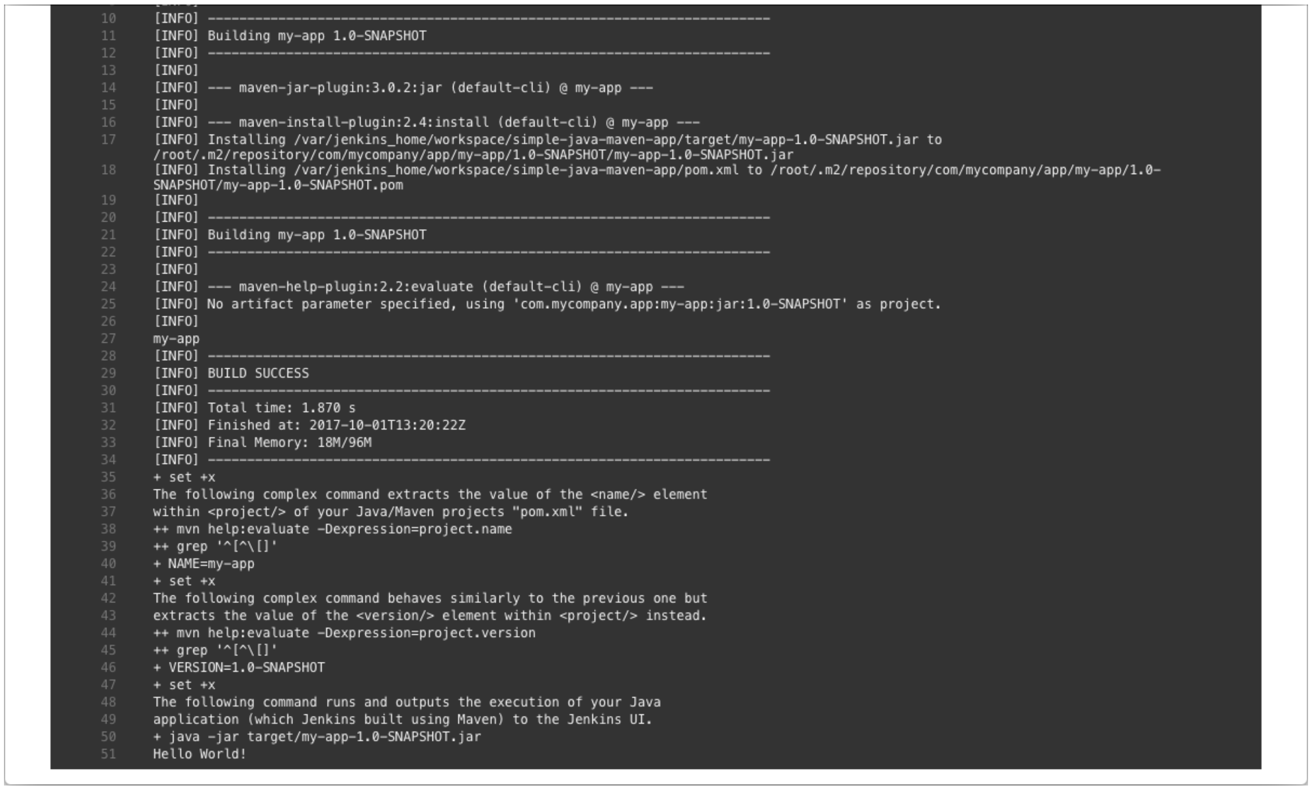
Defines a new stage called Deliver that appears on the Jenkins UI.

This sh step (of the steps section) runs the shell script deliver.sh located in the jenkins/scripts directory from the root of the simple-java-maven-app repository. Explanations about what this script does are covered in the deliver.sh file itself. As a general principle, it’s a good idea to keep your Pipeline code (i.e. the Jenkinsfile) as tidy as possible and place more complex build steps (particularly for stages consisting of 2 or more steps) into separate shell script files like the deliver.sh file. This ultimately makes maintaining your Pipeline code easier, especially if your Pipeline gains more complexity.

7. Save your edited Jenkinsfile and commit it to your local simple-java-maven-app Git repository. E.g. Within the simple-java-maven-app directory, run the commands:
   git stage .
   then
   git commit -m "Add 'Deliver' stage"
8. Go back to Jenkins again, log in again if necessary and ensure you've accessed Jenkins's Blue Ocean interface.
9. Click **Run** at the top left, then quickly click the **OPEN** link which appears briefly at the lower-right to see Jenkins running your amended Pipeline project. If you weren't able to click the **OPEN** link, click the *top* row on the Blue Ocean interface to access this feature.
   If your amended Pipeline ran successfully, here's what the Blue Ocean interface should look like. Notice the additional "Deliver" stage. Click on the previous "Test" and "Build" stage circles to access the outputs from those stages.

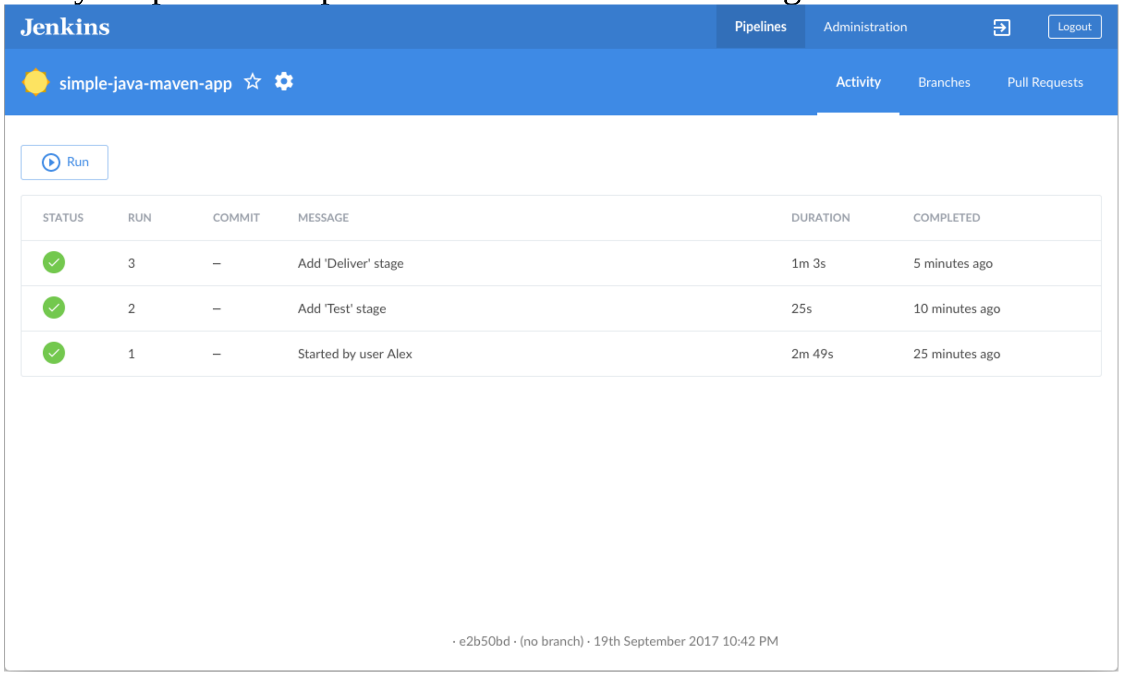


Here's what the output of the "Deliver" stage should look like, showing you the execution results of your Java application at the end.



```
[INFO] ------------------------------------------------------------------------
[INFO] Building my-app 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- maven-jar-plugin:3.0.2:jar (default-cli) @ my-app ---
[INFO]
[INFO] --- maven-install-plugin:2.4:install (default-cli) @ my-app ---
[INFO] Installing /var/jenkins_home/workspace/simple-java-maven-app/target/my-app-1.0-SNAPSHOT.jar to
/root/.m2/repository/com/mycompany/app/my-app/1.0-SNAPSHOT/my-app-1.0-SNAPSHOT.jar
[INFO] Installing /var/jenkins_home/workspace/simple-java-maven-app/pom.xml to /root/.m2/repository/com/mycompany/app/my-app/1.0-
SNAPSHOT/my-app-1.0-SNAPSHOT.pom
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building my-app 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- maven-help-plugin:2.2:evaluate (default-cli) @ my-app ---
[INFO] No artifact parameter specified, using 'com.mycompany.app:my-app:jar:1.0-SNAPSHOT' as project.
[INFO]
my-app
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1.870 s
[INFO] Finished at: 2017-10-01T13:20:22Z
[INFO] Final Memory: 18M/96M
[INFO] ------------------------------------------------------------------------
+ set +x
The following complex command extracts the value of the <name/> element
within <project/> of your Java/Maven projects "pom.xml" file.
++ mvn help:evaluate -Dexpression=project.name
++ grep '^[^\[]'
+ NAME=my-app
+ set +x
The following complex command behaves similarly to the previous one but
extracts the value of the <version/> element within <project/> instead.
++ mvn help:evaluate -Dexpression=project.version
++ grep '^[^\[]'
+ VERSION=1.0-SNAPSHOT
+ set +x
The following complex command runs and outputs the execution of your Java
application (which Jenkins built using Maven) to the Jenkins UI.
+ java -jar target/my-app-1.0-SNAPSHOT.jar
Hello World!
```

10. Click the **X** at the top-right to return to the main Blue Ocean interface, which lists your previous Pipeline runs in reverse chronological order.



STATUS	RUN	COMMIT	MESSAGE	DURATION	COMPLETED
✓	3	–	Add 'Deliver' stage	1m 3s	5 minutes ago
✓	2	–	Add 'Test' stage	25s	10 minutes ago
✓	1	–	Started by user Alex	2m 49s	25 minutes ago

· e2b50bd · (no branch) · 19th September 2017 10:42 PM

Wrapping up

Well done! You’ve just used Jenkins to build a simple Java application with Maven!

The "Build", "Test" and "Deliver" stages you created above are the basis for building more complex Java applications with Maven in Jenkins, as well as Java and Maven applications that integrate with other technology stacks.

Because Jenkins is extremely extensible, it can be modified and configured to handle practically any aspect of build orchestration and automation.

Conclusion

The application is successfully built in Docker, and hosted in Docker Hub

It is deployed on ELK stack on Docker and push application logs to it
and also I have madeAutomate Docker build and deployment using Jenkins pipeline code

By making use of following tools:

- Docker
- Docker Compose
- Elasticsearch
- Logstash
- Kibana
- Spring Boot application