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

Tools used:

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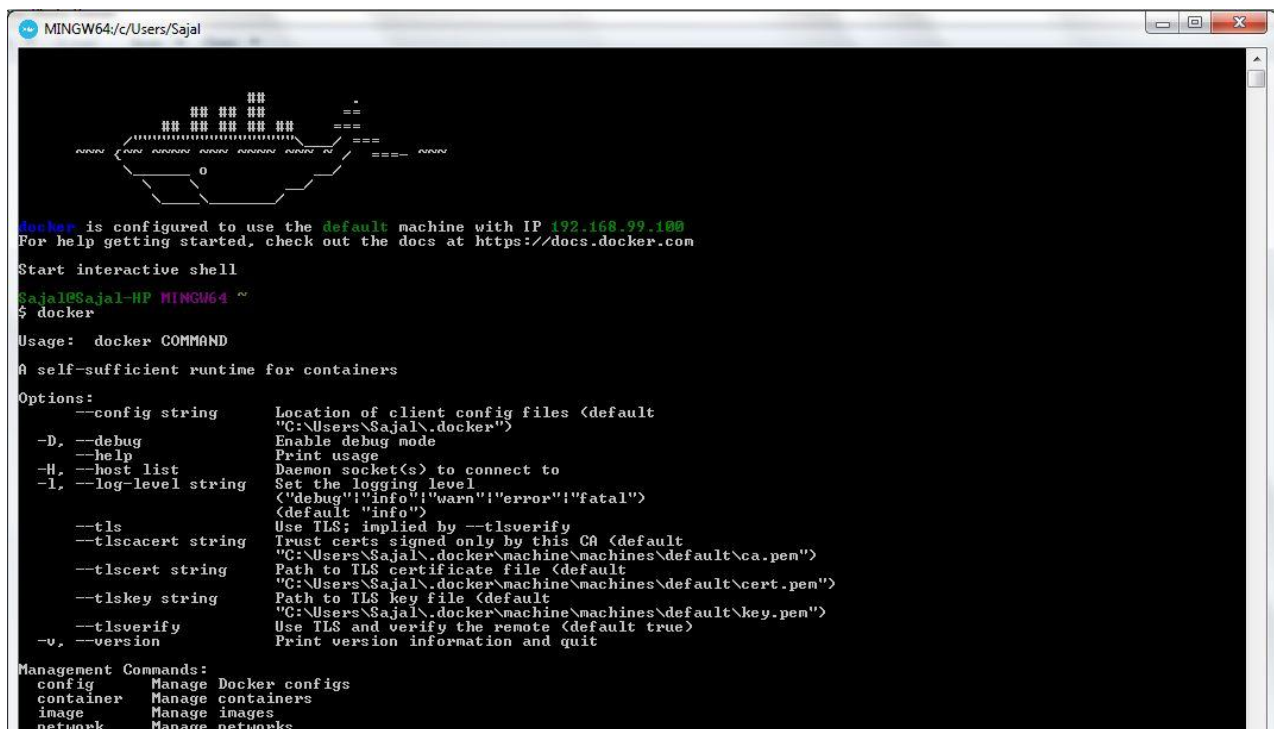
If we don't have this enabled then we need to go to the BIOS option and enable Hardware Virtualization. The BIOS is a bit different for different models of Computer, so please follow the official guideline for enabling that.

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.

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.

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.

```java
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>
```

```xml
<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.

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.

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>

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.

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.
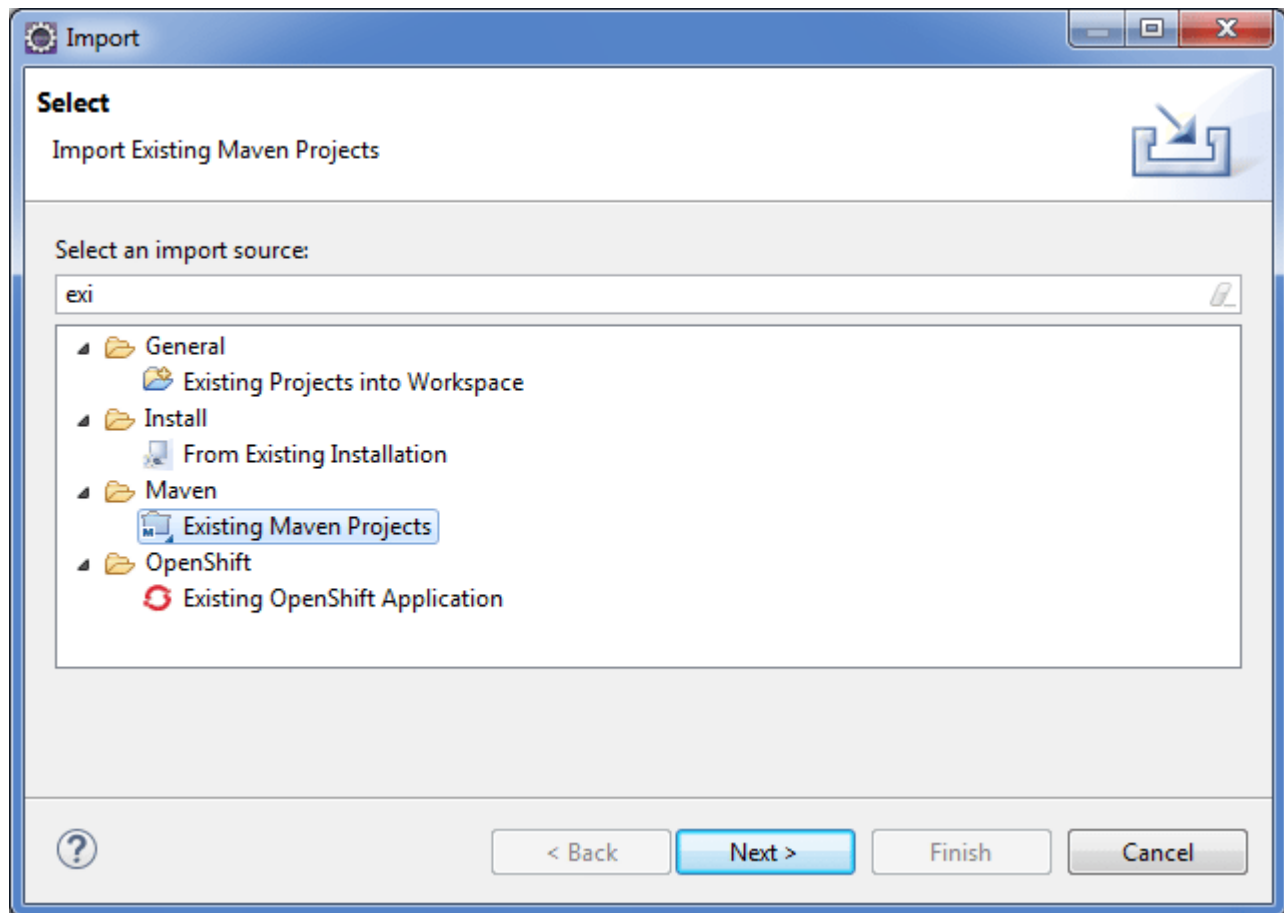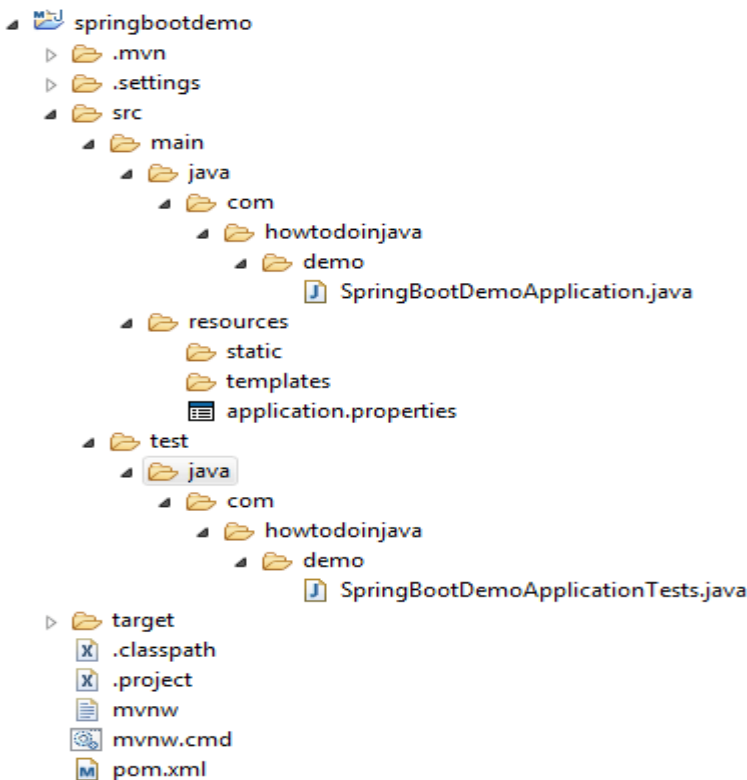
Import Existing Maven Project into Eclipse

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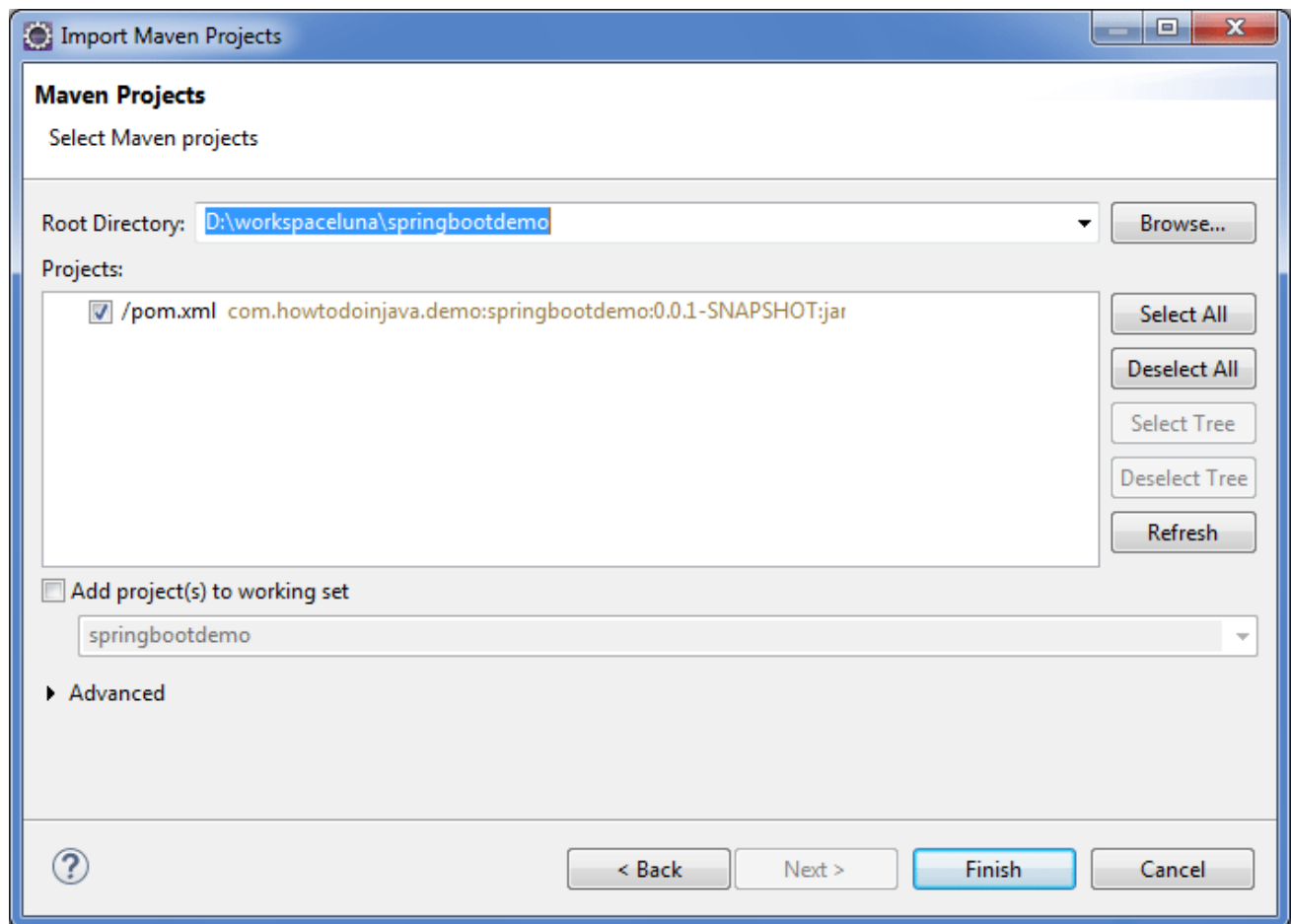
Select pom.xml file to import maven project

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.

<div align="center">SpringBootDemoApplication.java</div>

import java.util.Arrays;

```java
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.

<div align="center">Console</div>

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;

  }
```

```java
    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.

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;

```java
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;
    }
```

```java
@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})  %{LC
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.
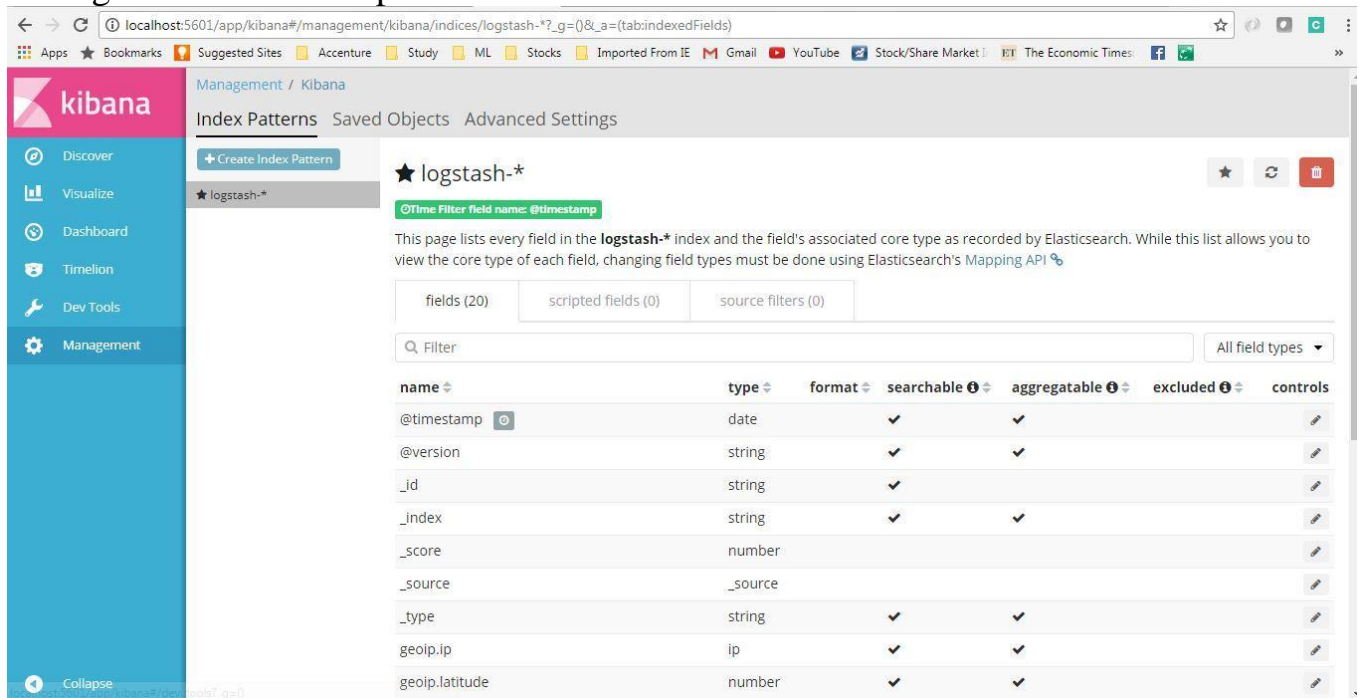
Logstash configuration in Kibana

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.
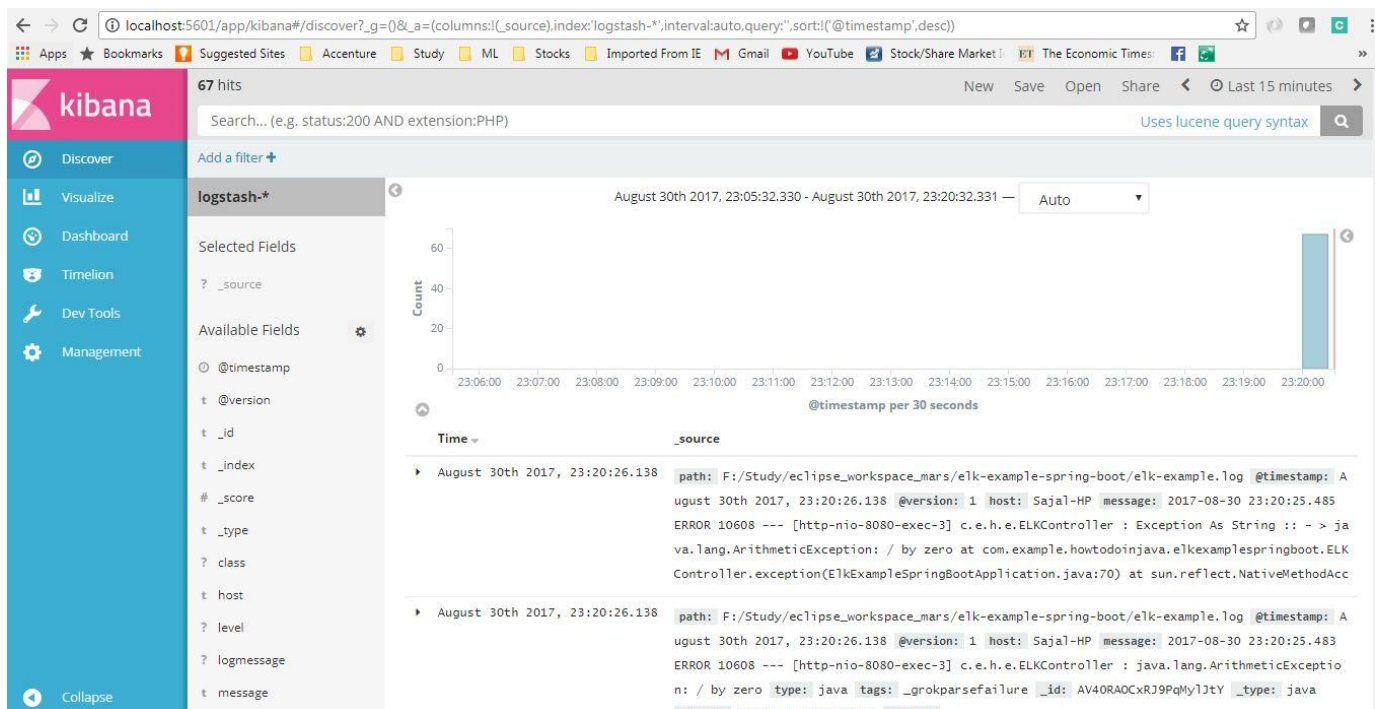
Kibana Logs Overview
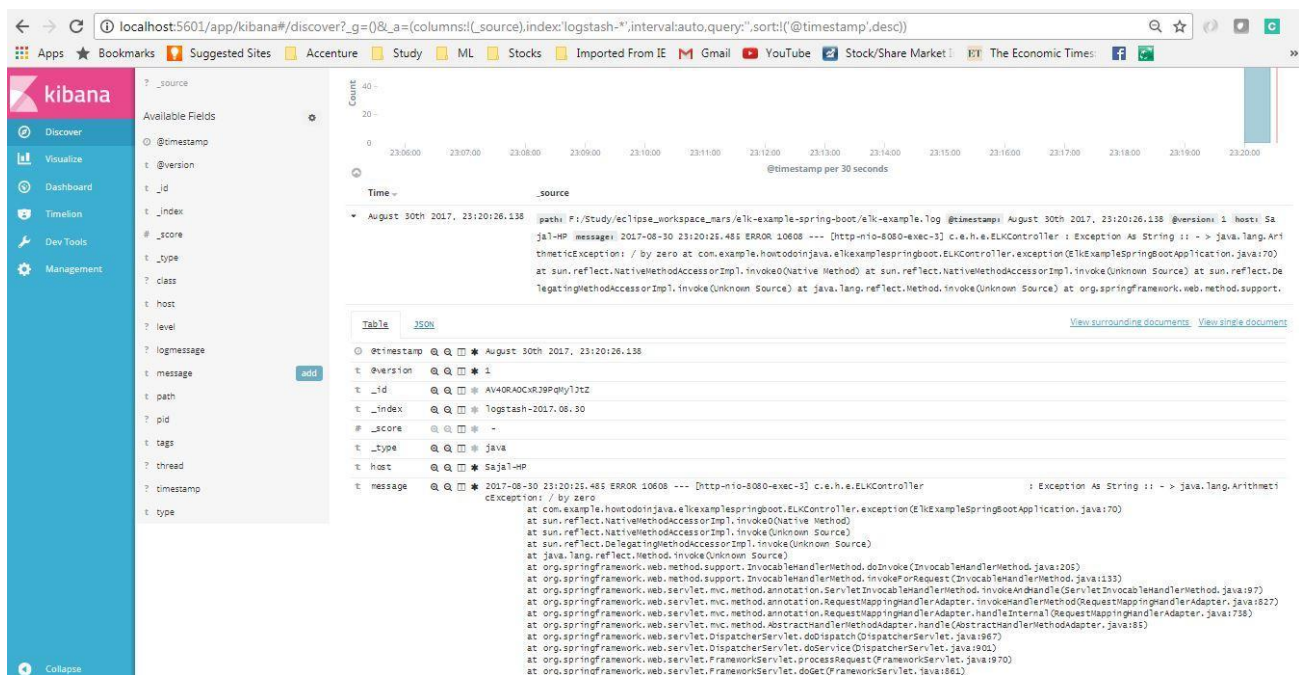
Kibana Logs details screen

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:

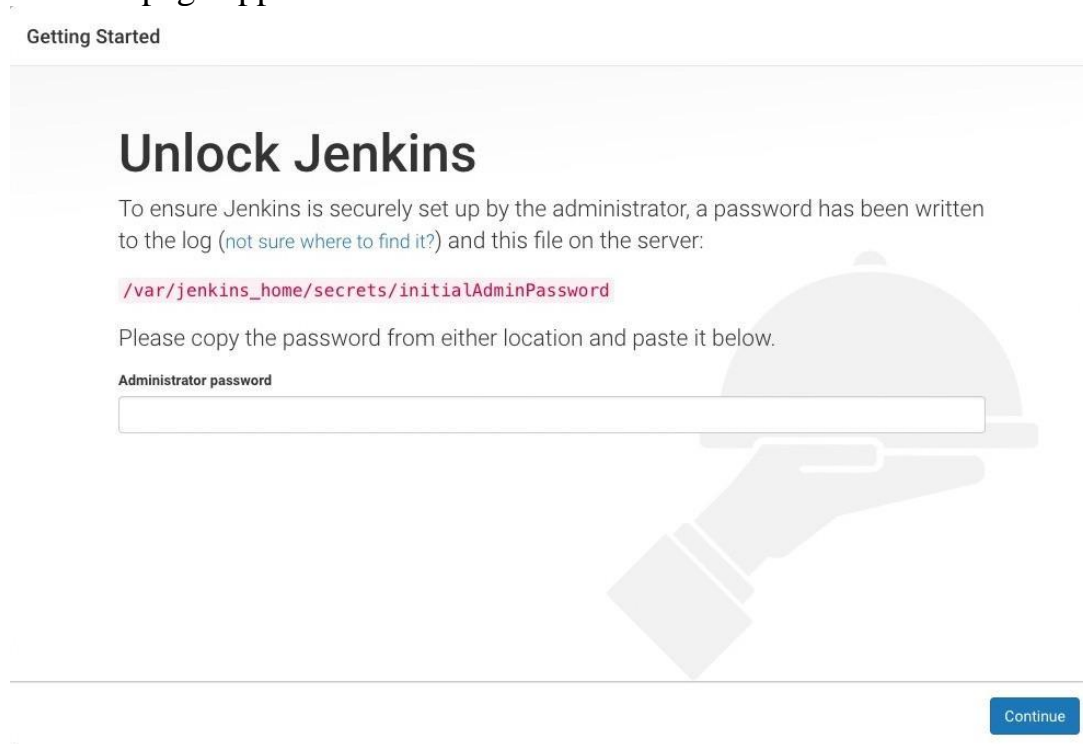docker container exec -it jenkins-blueocean bash

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.

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).

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

Customizing Jenkins with plugins

After <u>unlocking Jenkins</u>, 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:
   o This page may indicate **Jenkins is almost ready!** instead and if so, click **Restart**.

o 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):
   o 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 <u>above</u>, 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 <u>Pipeline</u> and <u>Using a Jenkinsfile</u> 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:
3. pipeline {
4.   agent {
5.   docker {
6.   image 'maven:3-alpine'
7.   args '-v /root/.m2:/root/.m2'
8.   }
9.   }
10.   stages {
11.   stage('Build') {
12.   steps {
13.   sh 'mvn -B -DskipTests clean package'
14.   }
15.   }
16.   }
    }

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:

- o You'll have separate Jenkins and Maven containers running locally in Docker.
- o 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:
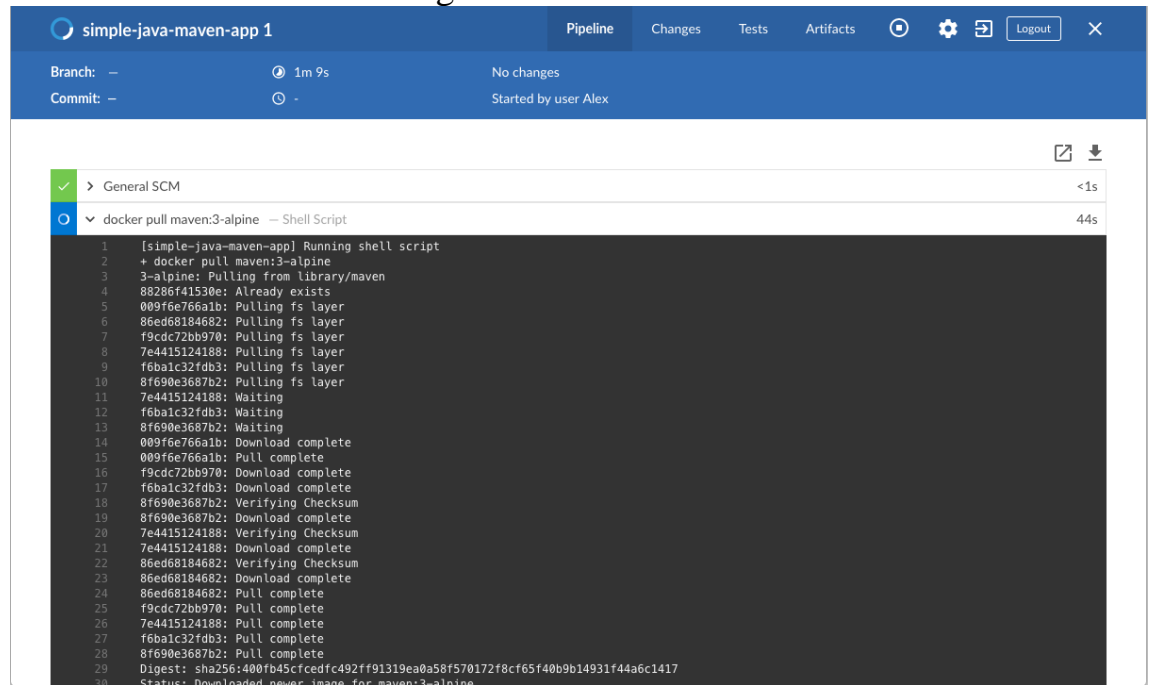git add .
then
git commit -m "Add initial Jenkinsfile"
18. Go back to Jenkins again, log in again if necessary and click **Open Blue Ocean** on the left to access Jenkins's Blue Ocean interface.

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.

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).

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.

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:

```
3.        stage('Test') {
4.            steps {
5.                sh 'mvn test'
6.            }
7.            post {
8.                always {
9.                    junit 'target/surefire-reports/*.xml'
10.                }
11.            }
        }
```

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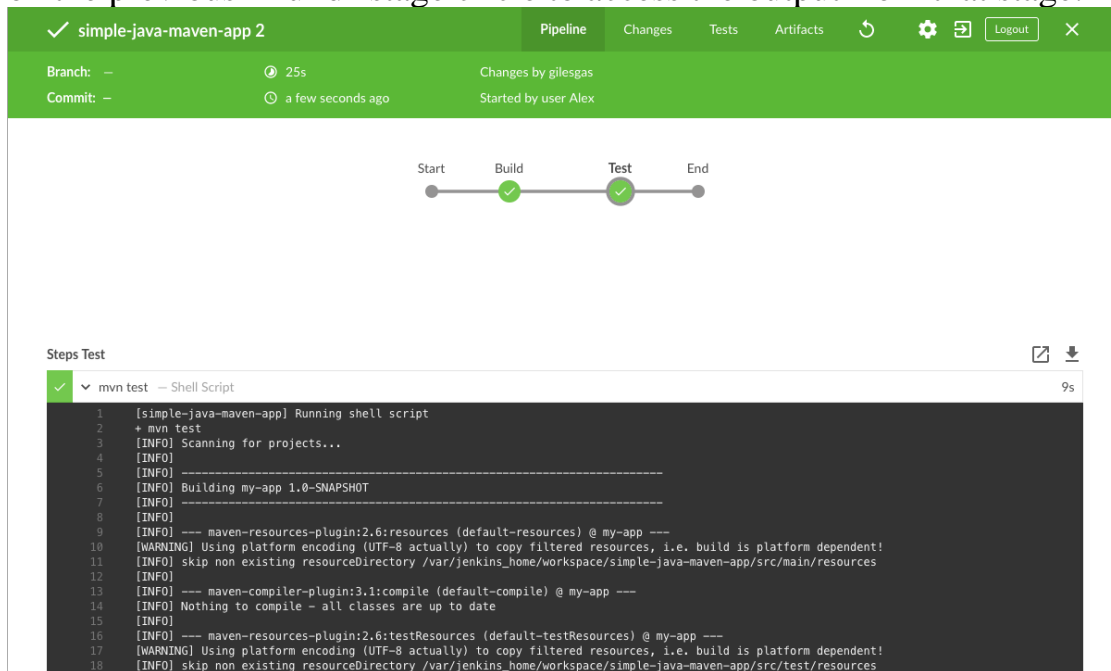
15. Click the **X** at the top-right to return to the main Blue Ocean interface.

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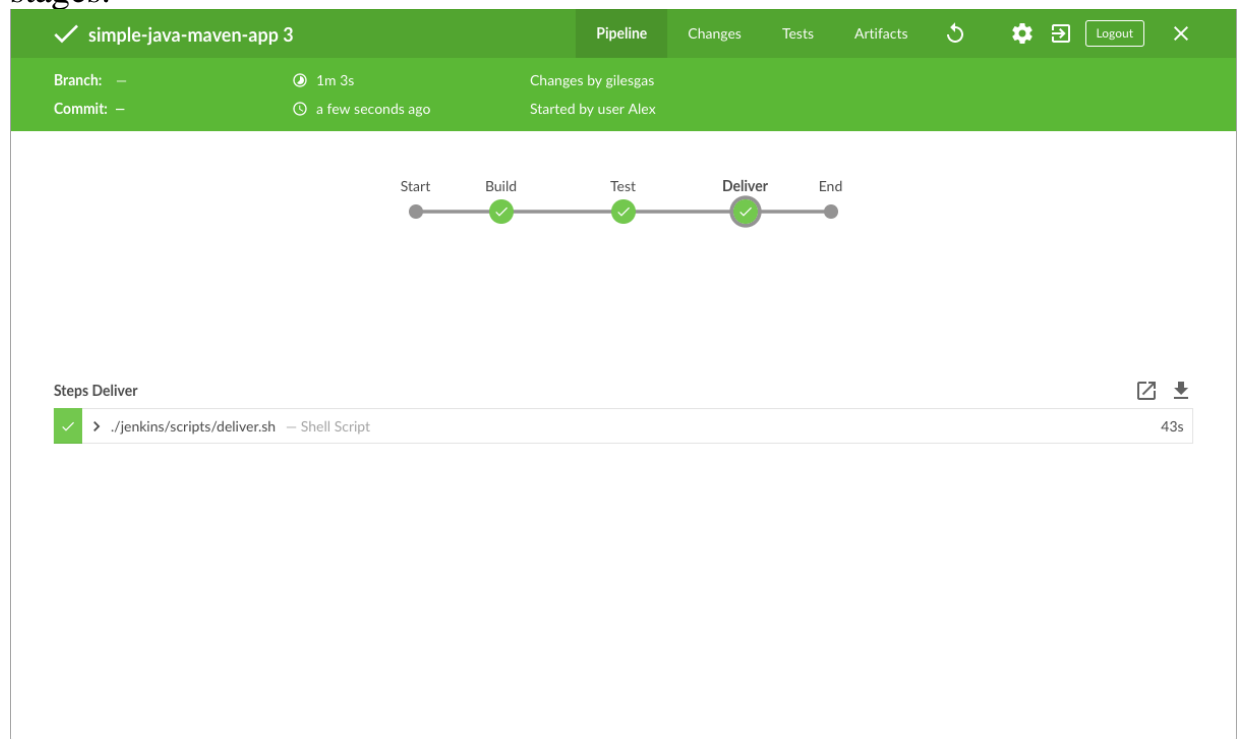
Here's what the output of the "Deliver" stage should look like, showing you the execution results of your Java application at the end.

```
10    [INFO] ------------------------------------------------------------------
11    [INFO] Building my-app 1.0-SNAPSHOT
12    [INFO] ------------------------------------------------------------------
13    [INFO]
14    [INFO] --- maven-jar-plugin:3.0.2:jar (default-cli) @ my-app ---
15    [INFO]
16    [INFO] --- maven-install-plugin:2.4:install (default-cli) @ my-app ---
17    [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
18    [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
19    [INFO]
20    [INFO] ------------------------------------------------------------------
21    [INFO] Building my-app 1.0-SNAPSHOT
22    [INFO] ------------------------------------------------------------------
23    [INFO]
24    [INFO] --- maven-help-plugin:2.2:evaluate (default-cli) @ my-app ---
25    [INFO] No artifact parameter specified, using 'com.mycompany.app:my-app:jar:1.0-SNAPSHOT' as project.
26    [INFO]
27    my-app
28    [INFO] ------------------------------------------------------------------
29    [INFO] BUILD SUCCESS
30    [INFO] ------------------------------------------------------------------
31    [INFO] Total time: 1.870 s
32    [INFO] Finished at: 2017-10-01T13:20:22Z
33    [INFO] Final Memory: 18M/96M
34    [INFO] ------------------------------------------------------------------
35    + set +x
36    The following complex command extracts the value of the <name/> element
37    within <project/> of your Java/Maven projects "pom.xml" file.
38    ++ mvn help:evaluate -Dexpression=project.name
39    ++ grep '^[^\[]'
40    + NAME=my-app
41    + set +x
42    The following complex command behaves similarly to the previous one but
43    extracts the value of the <version/> element within <project/> instead.
44    ++ mvn help:evaluate -Dexpression=project.version
45    ++ grep '^[^\[]'
46    + VERSION=1.0-SNAPSHOT
47    + set +x
48    The following command runs and outputs the execution of your Java
49    application (which Jenkins built using Maven) to the Jenkins UI.
50    + java -jar target/my-app-1.0-SNAPSHOT.jar
51    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.

Jenkins   Pipelines   Administration   Logout

🟡 simple-java-maven-app ☆ ⚙   Activity   Branches   Pull Requests

▶ Run

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