Basic Docker Commands

1 docker --version

docker -v

This command is used to get the current version of the docker

```
PS C:\> docker -v
Docker version 20.10.17, build 100c701
PS C:\> docker --version
Docker version 20.10.17, build 100c701
```

2 docker pull: Pull an image or a repository from a registry

Syntax:docker pull [OPTIONS] NAME[: TAG|@DIGEST]

To download an image or set of images (i.e. A Repository)

Ex: **docker pull hello-world**

```
PS C:\> docker pull hello-world
Using default tag: latest
latest: Pulling from library/hello-world
c1ec31eb5944: Pull complete
Digest: sha256:d211f485f2dd1dee407a80973c8f129f00d54604d2c90732e8e320e5038a0348
Status: Downloaded newer image for hello-world:latest
docker.io/library/hello-world:latest
```

3 docker run: This command is used to create a container from an image

Syntax : docker run [OPTIONS] IMAGE [COMMAND] [ARG...]

Example : docker run hello-world

```
PS C:\> docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
c1ec31eb5944: Pull complete
Digest: sha256:d211f485f2dd1dee407a80973c8f129f00d54604d2c90732e8e320e5038a0348
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.
```

1. **docker ps**This command is used to list all the containers

Syntax : **docker ps [OPTIONS]**

Example :**docker ps (lists currently running containers)**

```
PS C:\> docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
```

Command : docker ps -a -a or -all : Lists all containers that are running & stopped

```
PS C:\> docker ps -a
CONTAINER ID   IMAGE         COMMAND    CREATED         STATUS                     PORTS     NAMES
e966ebcc530a   hello-world   "/hello"   4 minutes ago   Exited (0) 4 minutes ago             unruf
fled_spence
```

2. **docker restart**

This command is used to restart one or more containers.

Syntax: docker restart [OPTIONS] CONTAINER [CONTAINER...]

Example:docker restart my_container

6 docker kill

This command is used to kill one or more containers.

Syntax: docker kill [OPTIONS] CONTAINER [CONTAINER...]

Example:$ docker kill my_container

7Build an Image from a Dockerfile

docker build -t <image_name>

Ex: docker build -t

8 List local images

docker images

```
PS C:\> docker images
REPOSITORY              TAG       IMAGE ID       CREATED         SIZE
hello-world             latest    d2c94e258dcb   17 months ago   13.3kB
docker/getting-started  latest    3e4394f6b72f   22 months ago   47MB
```

9docker search To search for an image in Docker Hub.

docker search <image_name>

Ex: **docker search hello-world**

10Delete an Image

docker rmi <image_name>

Ex: **docker rmi hello-world**

Login into Docker

docker login -u <username>

Publish an image to Docker Hub

docker push <username>/<image_name>

Container commands

Create and run a container from an image, with a custom name:

docker run --name <container_name><image_name>

Run a container with and publish a container's port(s) to the host.

docker run -p <host_port>:<container_port><image_name>

Run a container in the background

docker run -d <image_name>

Start or stop an existing container:

docker start|stop <container_name> (or <container-id>)

Remove a stopped container:

docker rm <container_name>

Week 7: Create a Simple Java Containerized Application using Docker

Choose an application:

Choose a simple application that you want to containerize. Forexample, a Python script or any Java program

Here we use simple Java program and containerize it using docker

1 Create an application folder

2. Open the Visual Studio Code and select Open Folder from File menu

3. Now select the application folder that is created and open

4. From the explorer bar, click on the application folder

5. Now click on NewFile button to create a file

Now ,Create a simple Java program as below

```
public class Test
{
public static void main(String[] args)
{
int a=20,b=30,c;
c=a*b;
System.out.println("Hello world");
System.out.println("value of c is "+c);
}
}
```

Save it as with .java extension

Now again create another file and name the file as Dockerfile

Create a Dockerfile for the above Java program

In the Dockerfile, **specify the base image, copy the application into thecontainer, and specify the command to run the application**

Dockerfile

FROM openjdk # uses the base image –openjdk with java installed

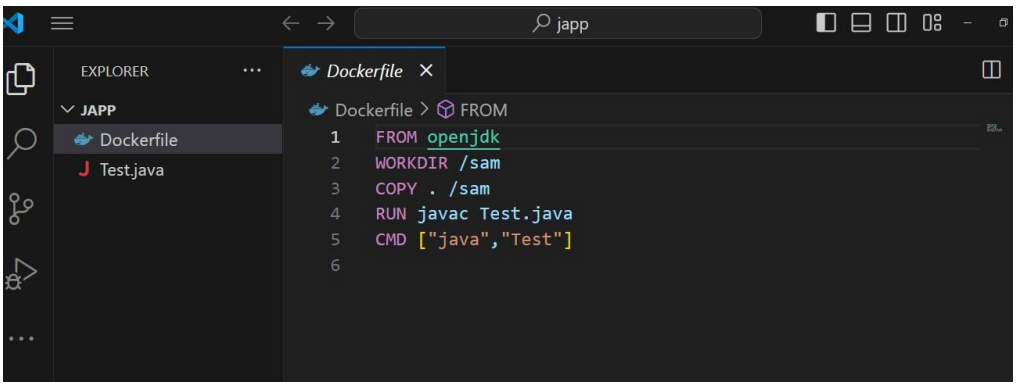
WORKDIR /sam# Set the working directory

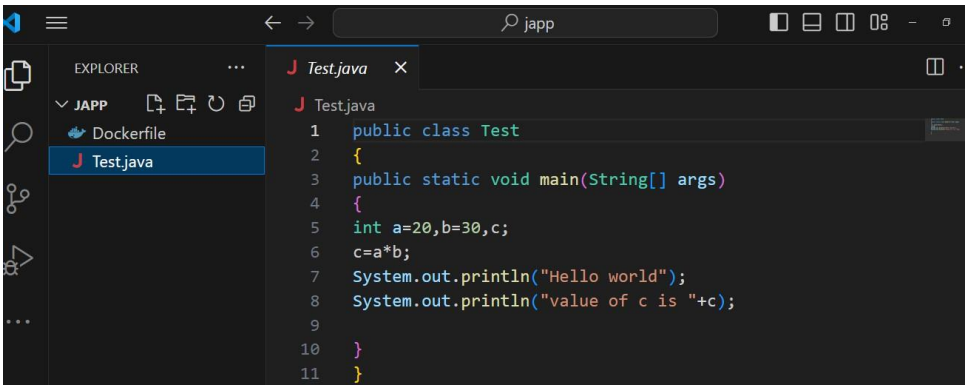
COPY. /sam# Copies all the files to working directory

RUN javac Test.java# Compile the Java program

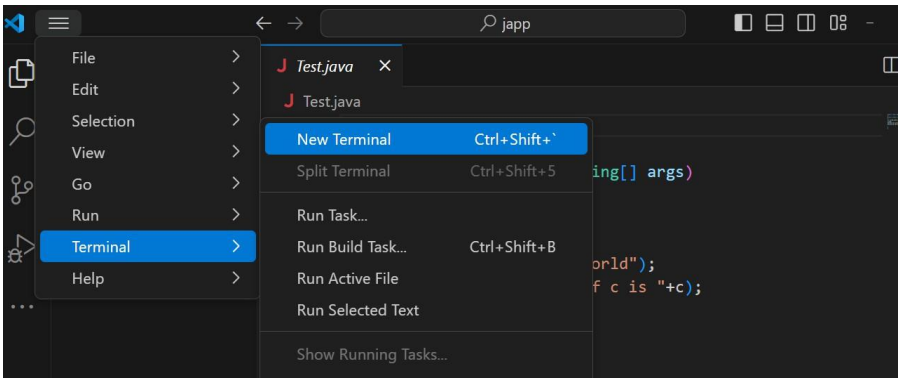
CMD ["java", "Test"]# Set the default command to run the Java program

Save the Java file and Docker file

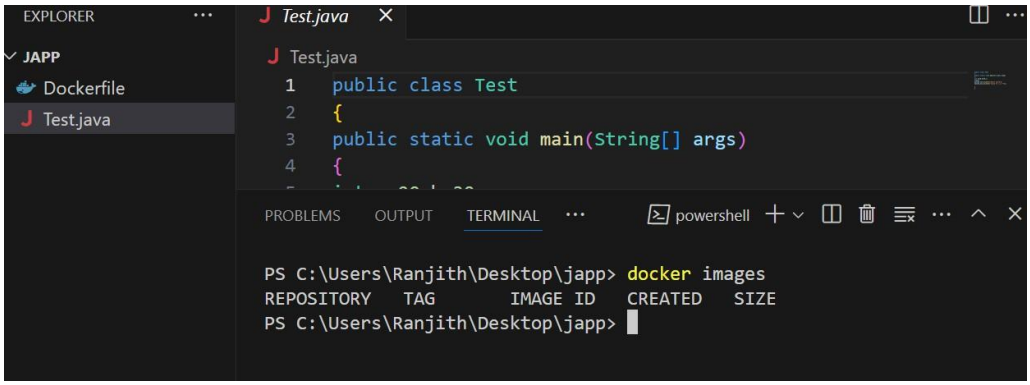




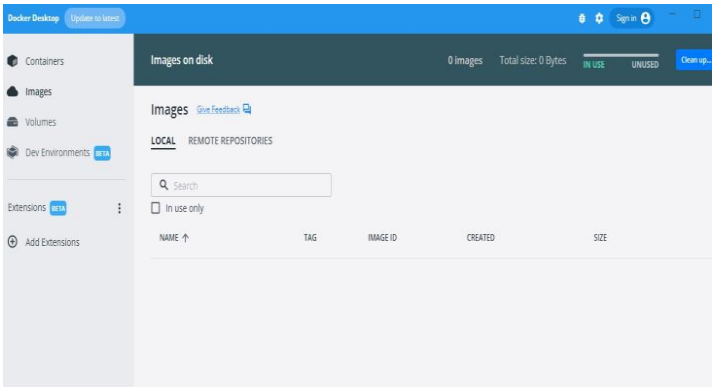
Start the New Terminal from Visual studio



Type the command to list all the images ::::**docker images**



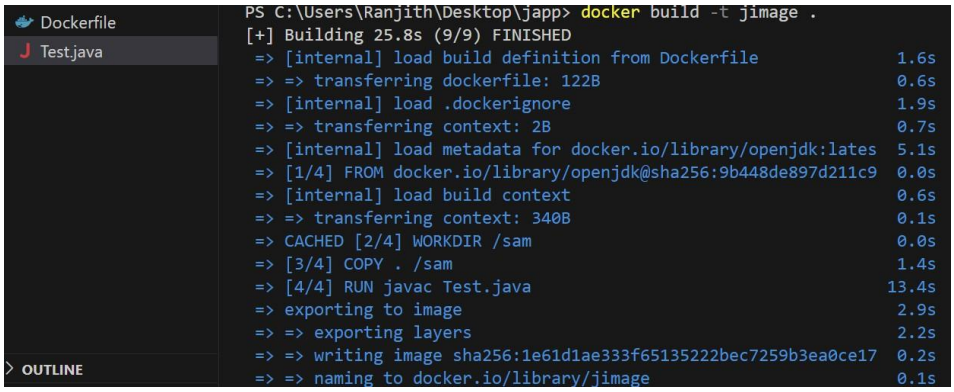
Start the Dockerdesktop



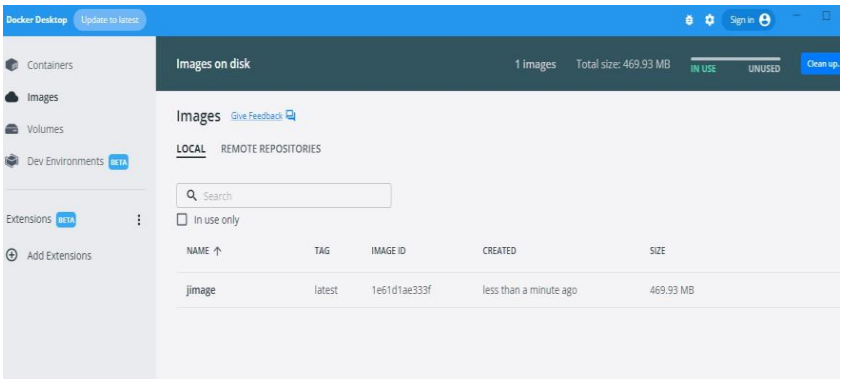
Build the Docker image using the **docker build** command. This command builds a new Docker image using the Dockerfile and tags theimage with the name

Command :docker build -t "any name for your docker image" .

Example :**docker build -t helloworld .**



View the image created in the docker



Run the Docker container:

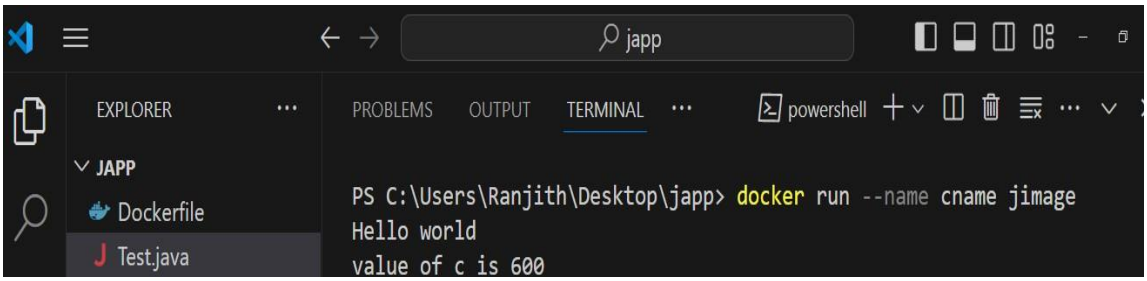
Run the following command to start a new container based on the image:

$ docker run --name mycontainermyimage

This command starts a new container named "mycontainer" based on the

"myimage" image and runs the Java code inside the container.

Example

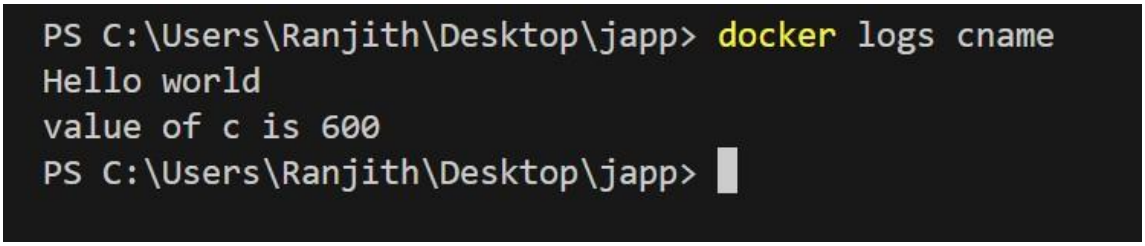


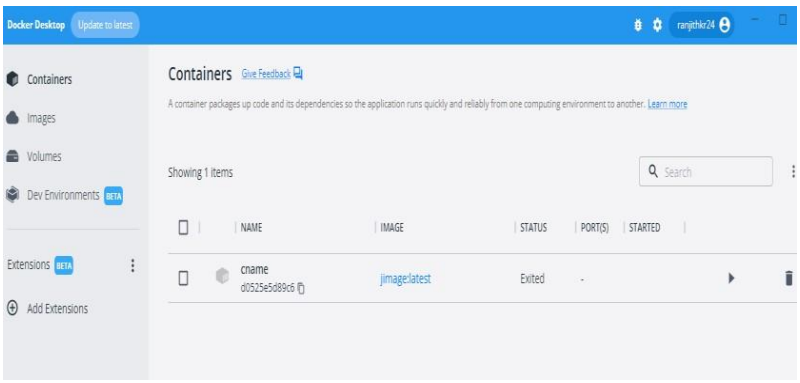
- **Verify the output**:

Run the following command to verify the output of the container:

$ docker logs mycontainer

This command displays the logs of the container and should show the programs output





Pushing images into the docker hub

Create a repository

Login to Dockerhub,Select Create Repository, name it and choose it to make itas public or private

Tag the Imageuse the command

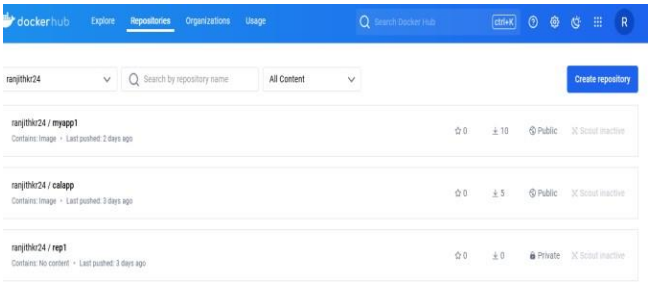
docker tag <image name><dockerhub-user name>/<repo-name><version>

Log into Docker

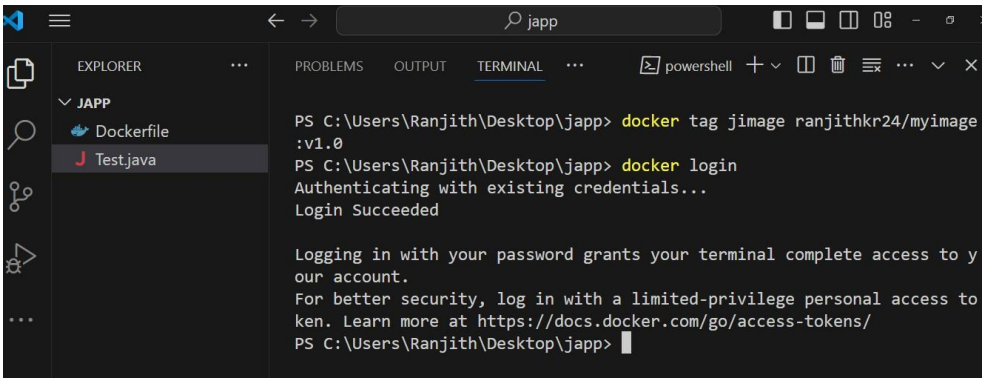
docker login **give login credentials**

Push the Imageuse the command **docker push <docker-hub-username>/<image name><version name>**

Login into Docker hub



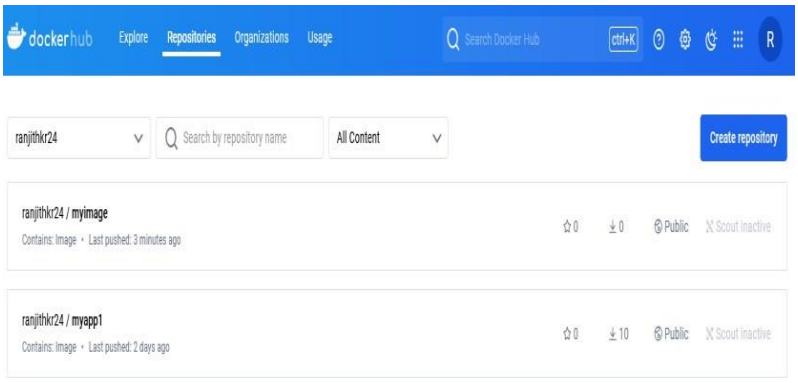
Tagging docker image and login into docker hub



Issuing docker push command to push the image

```
Logging in with your password grants your terminal complete access to y
our account.
For better security, log in with a limited-privilege personal access to
ken. Learn more at https://docs.docker.com/go/access-tokens/
PS C:\Users\Ranjith\Desktop\japp> docker push ranjithkr24/myimage:v1.0
The push refers to repository [docker.io/ranjithkr24/myimage]
d9081b57f2d9: Pushed
7f43ae0d9720: Pushed
020c055d83f3: Mounted from ranjithkr24/myapp1
56285d9a7760: Mounted from ranjithkr24/myapp1
077bff59ce57: Mounted from ranjithkr24/myapp1
9cd9df9ffc97: Mounted from ranjithkr24/myapp1
v1.0: digest: sha256:3d3de30bf1a86b342f65536c3122ff6552c015beaeb171e763
7d2d6d2342634f size: 1575
```

Pushed image can be viewed in dockerhub



Week 8: Integrate Kubernetes with Docker

Aim:

To enable Kubernetes on Docker Desktop and prepare system to run Kubernetes commands.

Kubernetes

Kubernetes, also known as K8s, is an **open-source system for automatingdeployment, scaling, and management of container-based applications** in different kinds of environments like physical, virtual, and cloud-native computing foundations.

Some of the benefits of Kubernetes

Container Orchestration: Kubernetes manages the placement and scheduling of containers across a cluster of machines, ensuring that the desired state of the application is maintained.

Scalability: Kubernetes allows applications to scale horizontally by automatically adding or removing containers based on resource utilization and user-defined rules.

Self-Healing: Kubernetes monitors the health of containers. It automatically restarts failed containers or replaces them with new ones to ensure the desired state of the application is maintained.

Rolling Updates and Rollbacks: Kubernetes supports rolling updates, allowing applications to be updated with minimal downtime. If any issues arise, it also facilitates rollbacks to previous versions.

Storage Orchestration: Kubernetes provides a way to manage persistent storage volumes and attach them to containers as needed.

Differences between Docker and Kubernetes

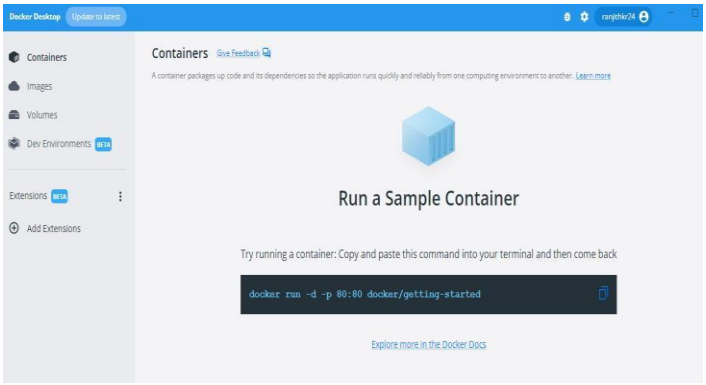
Docker is used for **building, shipping, and running containers**—it provides the means to create isolated environments for applications.

On the other hand, Kubernetes focuses on orchestrating containers, meaning it helps **manage, scale, and ensure the smooth operation of large collections of containers**.

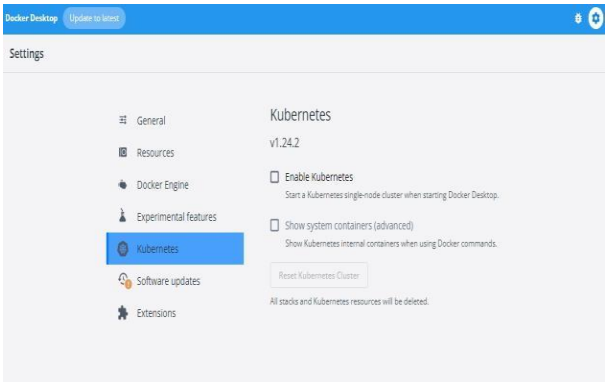
Docker manages individual containers, whereas Kubernetes manages multiple containers across clusters.

Integrating Kubernetes with Docker

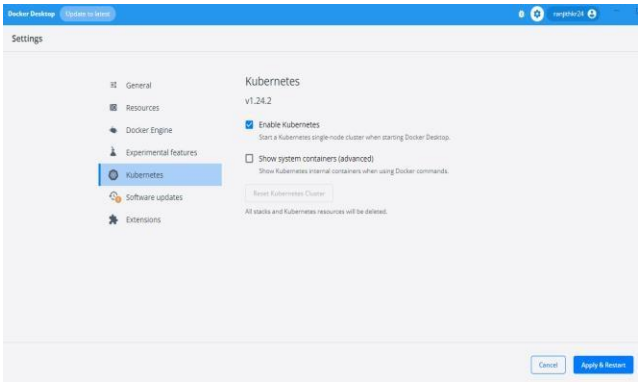
1. **Start the Docker Desktop**



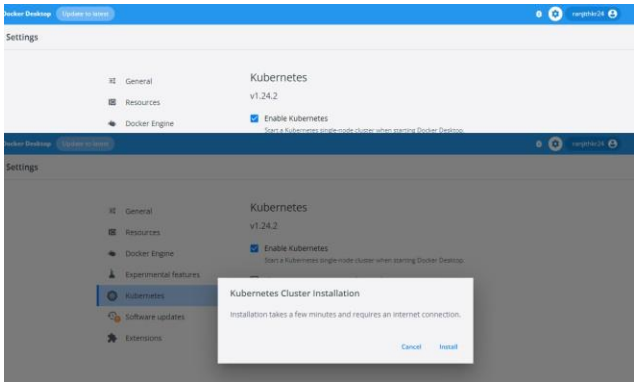
2. **Select Settings on the top right side from the Docker Desktop Dashboard**



3. Select Kubernetes from the left side bar
4. Check the Enable Kubernetes checkbox



5. Select Apply & restart to save the settings
6. Select Install to confirm



7.Go to Kubernetes website ,https://kubernetes.io/releases/download/#binaries

Download **kubectlconvert.exe for Windows OS**

8 Save this file **kubectlconvert.exe**in a folder**and copy the folder into**

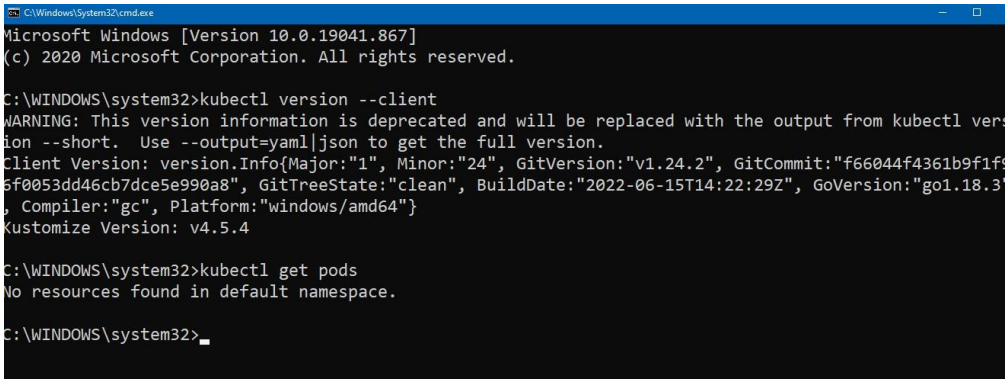
C//Windows//System32

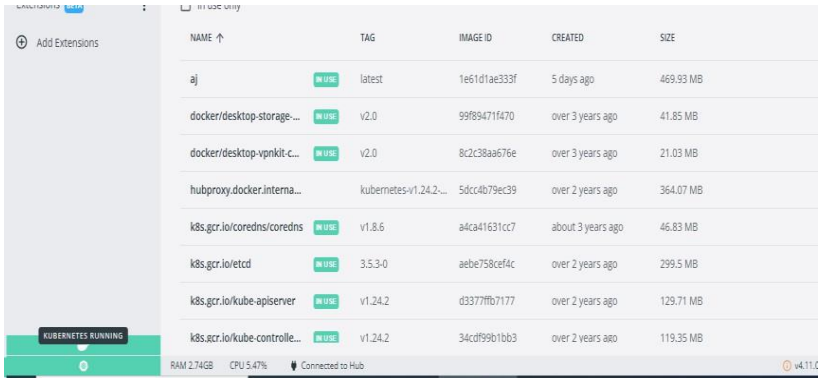
9. Set the path in environment path variable **for the kubectlconvert.exe file**

10. Open command prompt and type the following commands

kubectl version –client

kubectl get pods




Add Extensions
NAME
TAG
IMAGE ID
CREATED
SIZE
aj
IN USE
latest
1e61d1ae333f
5 days ago
469.93 MB
docker/desktop-storage-...
IN USE
v2.0
99f89471f470
over 3 years ago
41.85 MB
docker/desktop-vpnkit-c...
IN USE
v2.0
8c2c38aa676e
over 3 years ago
21.03 MB
hubproxy.docker.interna...
kubernetes-v1.24.2-...
5dcc4b79ec39
over 2 years ago
364.07 MB
k8s.gcr.io/coredns/coredns
IN USE
v1.8.6
a4ca41631cc7
about 3 years ago
46.83 MB
k8s.gcr.io/etcd
IN USE
3.5.3-0
aebe758cef4c
over 2 years ago
299.5 MB
k8s.gcr.io/kube-apiserver
IN USE
v1.24.2
d3377ffb7177
over 2 years ago
129.71 MB
k8s.gcr.io/kube-controlle...
IN USE
v1.24.2
34cdf99b1bb3
over 2 years ago
119.35 MB
KUBERNETES RUNNING
RAM 2.74GB
CPU 5.47%
Connected to Hub

Week 9: Automate the Running of a Containerized Application

Kubernetes is an open-source platform for **managing containerized workloads and services**. It provides a **robust framework for automating deployments, scaling, and managing containerized applications**

Prerequisites

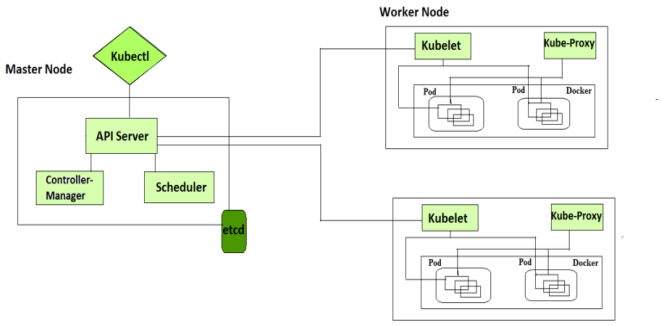
following prerequisites are required:

- **Kubernetes Cluster**: A running Kubernetes cluster. You can set up a local cluster using Minikube or use a managed service like Google Kubernetes Engine (GKE).
- **kubectl**: The command-line tool for interacting with your Kubernetes cluster.
- **Docker**: Installed and running on your machine.
- **Containerized Application**: Your application packaged into a Docker image.

Kubernetes Cluster Architecture

Kubernetes comes with a client-server architecture. It consists of master and worker nodes

The master node, contains the components such as API Server, controller manager, scheduler, and etcddatabase for stage storage. kubelet to communicate with the master, the kube-proxy for networking, and a container runtime such as Docker to manage containers.



Kubernetes Components

Kubernetes is composed of a number of components, each of which plays a specific role in the overall system. These components can be divided into two categories:

- **nodes:** Each Kubernetes cluster requires at least one worker node, which is a collection of worker machines that make up the nodes where our container will be deployed.
- **Control plane:** The worker nodes and any pods contained within them will be under the control plane.

Control Plane Components

It is basically a collection of various components that help us in managing the overall health of a cluster. For example, if you want to set up new pods, destroy pods, scale pods, etc. Basically, 4 services run on Control Plane**:**

1. Kube-API server

2. Kube-Scheduler

3. Kube-Controller-Manager

4. etcd

Node Components

These are the nodes where the actual work happens. Each Node can have multiple pods and pods have containers running inside them.

1. Container runtime

2. kubelet

3. kube-proxy

Test.java

```
import java.io.IOException;

import java.net.InetSocketAddress;

import com.sun.net.httpserver.HttpServer;
```

```
public class Test {

   public static void main(String[] args) throws IOException

{

      int port = 8080; // Port on which the server will listen

      System.out.println("Starting server on port " + port);

      HttpServer server = HttpServer.create(new InetSocketAddress(port), 0);

      server.createContext("/", (exchange -> {

         String response = "Hello, Kubernetes!";

         exchange.sendResponseHeaders(200, response.getBytes().length);

         exchange.getResponseBody().write(response.getBytes());

         exchange.close();

      }));

      server.setExecutor(null);

      server.start();

      System.out.println("Server is running at http://localhost:" + port);

   }

}
```

Step 1: Create a Docker Image

First, create a Docker image for your application. This involves writing a Dockerfile that specifies the base image, copies the application code, and sets the command to run the application.

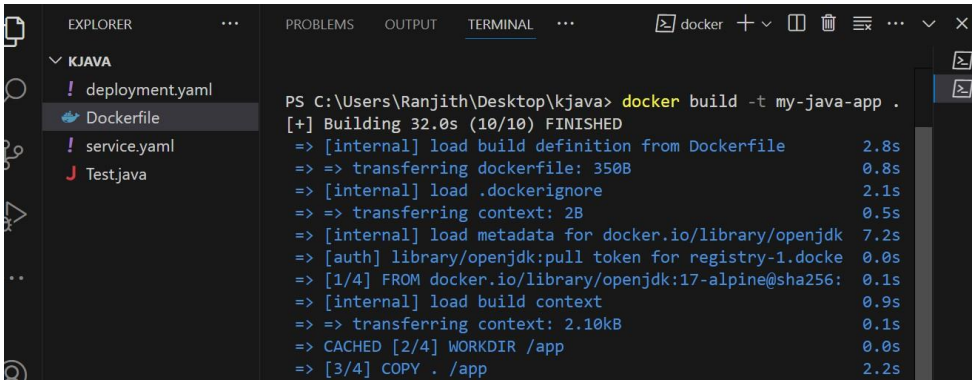
Dockerfile

```
# Use OpenJDK base image

FROM openjdk:17-alpine
```

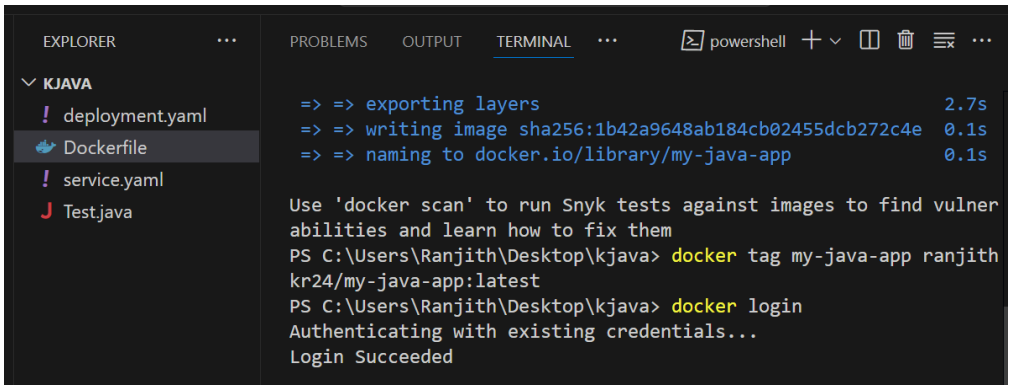
```
# Set working directory

WORKDIR /app

# Copy all files to the working directory

COPY . /app

# Compile the Java application

RUN javac Test.java

# Expose port 8080 for the application

EXPOSE 8080

# Run the Java application

CMD ["java", "Test"]
```

Build the Docker image using the Dockerfile:

- **docker build -t my-java-app** .



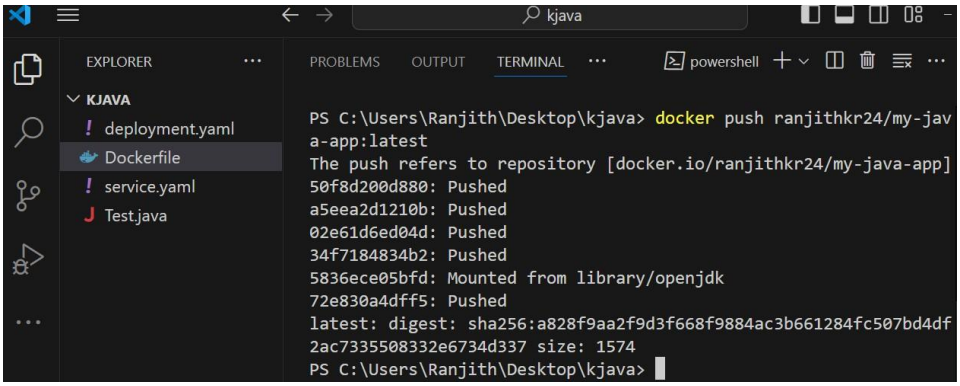
Tagging the image and login into dockerhub



Step 2: Push the Docker Image to a Registry

Push the Docker image to a container registry like Docker Hub

docker tag my-java-app:latest<your-docker-hub-username>/my-java-app:latest
docker push <your-docker-hub-username>/my-java-app:latest



Step 3: Create a Kubernetes Deployment

A Kubernetes Deployment is a resource that manages the rollout of new versions of an application. It ensures that a specified number of replicas (i.e., copies) of your application are running at any given time.

Create a YAML file named deployment.yaml with the following content:

apiVersion: apps/v1

kind: Deployment

metadata:

```
  name: my-java-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-java-app
  template:
    metadata:
      labels:
        app: my-java-app
    spec:
      containers:
        - name: my-java-app
          image: ranjithkr24/my-java-app:latest
          ports:
            - containerPort: 8080
          env:
            - name: PORT
              value: "8080"
```

Apply the deployment configuration to your Kubernetes cluster:

kubectl apply -f deployment.yaml

Step 4: Create a Kubernetes Service

A Kubernetes Service provides a network identity and load balancing for accessing your application. Create a YAML file named service.yaml with the following content:

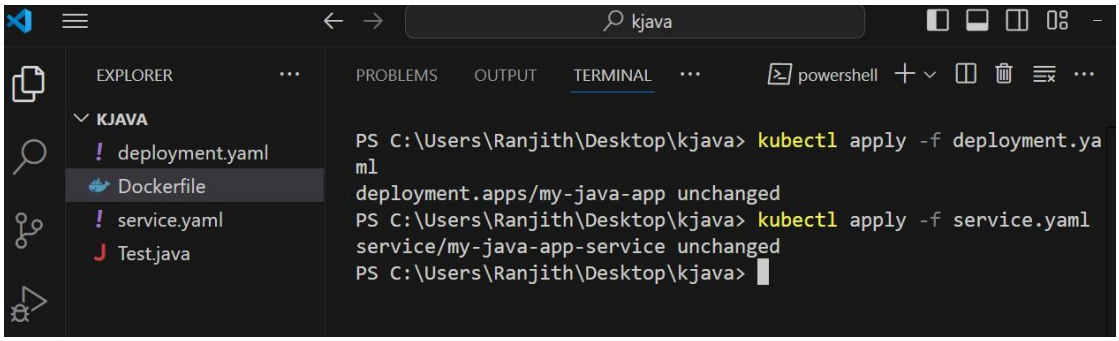
apiVersion: v1

kind: Service

```
metadata:
  name: my-java-app-service
spec:
  selector:
    app: my-java-app
  ports:
    - protocol: TCP
      port: 80        # Port exposed inside the cluster
      targetPort: 8080  # Port your Java app is running on
      nodePort: 30007   # External port
  type: NodePort
```

Apply the service configuration to your Kubernetes cluster:

kubectl apply -f service.yaml



Step 5: Verify the Deployment

Check the status of your deployment and service:

kubectl get deployments

```
PS C:\Users\Ranjith\Desktop\kjava> kubectl get deployments
NAME              READY   UP-TO-DATE   AVAILABLE   AGE
deployment1       0/3     3            0           7d5h
java-app          0/3     3            0           45h
java-app1         0/3     1            0           45h
java-deployment   0/3     3            0           7d14h
my-java-app       2/3     3            2           46h
myjava1-deploy    0/3     3            0           8d
PS C:\Users\Ranjith\Desktop\kjava> kubectl get deployments
NAME              READY   UP-TO-DATE   AVAILABLE   AGE
deployment1       0/3     3            0           7d5h
java-app          0/3     3            0           45h
java-app1         0/3     1            0           45h
java-deployment   0/3     3            0           7d14h
my-java-app       3/3     3            3           46h
myjava1-deploy    0/3     3            0           8d
```

kubectl get pods

```
Dockerfile
! service.yaml
J Test.java
(4m1s ago)    7d14h
java-deployment-647885494-v2fv7   0/1     CrashLoopBackOff   239
(3m50s ago)   7d14h
my-java-app-5d79d95f68-bwhm4      1/1     Running            2 (4
2m ago)       46h
my-java-app-5d79d95f68-bz42k      1/1     Running            3
              46h
my-java-app-5d79d95f68-mcjjr      1/1     Running            2 (4
2m ago)       46h
```

kubectl get svc

```
KJAVA
! deployment.yaml
Dockerfile
! service.yaml
J Test.java

PS C:\Users\Ranjith\Desktop\kjava> kubectl get svc
NAME                 TYPE           CLUSTER-IP       EXTERNAL-IP
   PORT(S)        AGE
java-app-service     NodePort       10.96.60.200     <none>
   80:30008/TCP   45h
java-app1-service    NodePort       10.107.54.172    <none>
   80:30010/TCP   45h
java-service         LoadBalancer   10.96.207.224    localhost
   80:32029/TCP   7d15h
kubernetes           ClusterIP      10.96.0.1        <none>
   443/TCP        11d
my-java-app-service  NodePort       10.106.150.206   <none>
   80:30007/TCP   46h
myjava1-service      LoadBalancer   10.103.169.241   localhost
   80:31676/TCP   8d
```

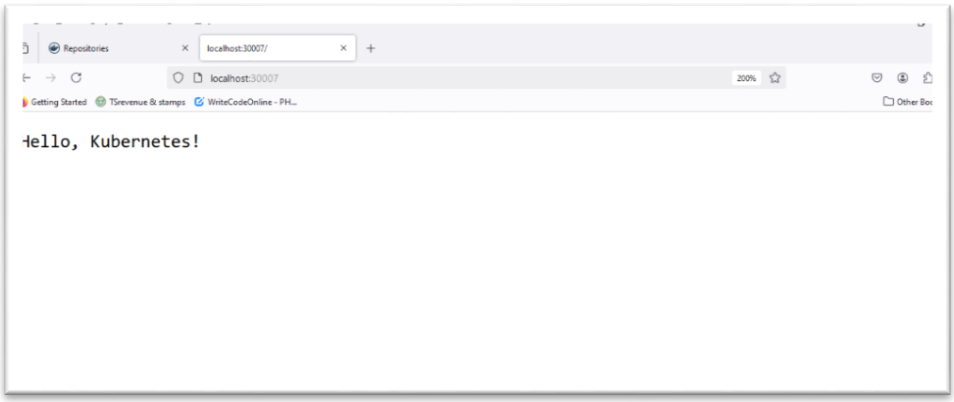
Step 6: Access the Application

To access your application, you need to get the external IP address of the service:

Open a web browser and navigate to the external IP address to access your application.

Type: http://localhost:port number of the application service

Type in the browser -- http://localhost:30007 to access the application for the above example



10 Install and Explore Selenium for automated testin

What is Selenium?

Selenium is an open-source tool used for automating web browsers. It provides a suite of tools for testing web applications across different browsers and platforms. The Selenium WebDriver API allows you to interact with web pages programmatically by simulating user actions like clicking, typing, navigating, etc.

Installation of Selenium WebDriver

Step 1: Install Node.js and npm

```
D:\py54\Java_test\Java_test>npm -v
10.2.5
```

If you don't have Node.js installed, download it from Node.js official website. Follow the instructions for your operating system.

1. **Verify the installation by running the following commands in the terminal:**

node -v

npm -v

```
D:\py54\Java_test\Java_test>node -v
v20.10.0
```

```
D:\>mkdir jtest

D:\>cd jtest
```

Step 2: Create a Project Folder

1. **Open Visual Studio Code.**
2. **Create a new folder named SeleniumLab.**

3. **Open the terminal in VS Code and initialize a Node.js project:**

npm init -y

```
D:\jtest>npm init -y
Wrote to D:\jtest\package.json:
```

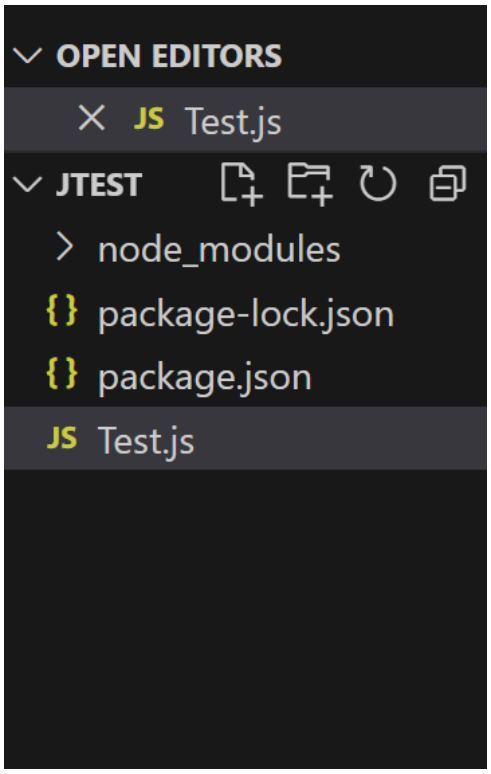
Step 3: Install Selenium WebDriver

1. **Install the selenium-webdriver package:**

npm install selenium-webdriver

```
D:\jtest>npm install selenium-webdriver
```

2. **Download the ChromeDriver that matches your Chrome browser version from ChromeDriver Downloads.**
3. **Extract the downloaded ChromeDriver and place it in the project folder.**

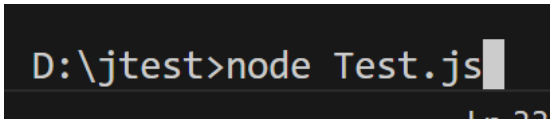


Part 3: Writing a Simple Selenium Test Script

Step 1: Create a JavaScript Test File

1. **Inside the SeleniumLab folder, create a new file named test.js.**
2. **Copy and paste the following code into test.js:**

Test.js




```
const { Builder, By, Key, until } = require('selenium-webdriver');

async function testGoogleSearch() {

  // Step 1: Launch the Chrome browser

  let driver = await new Builder().forBrowser('chrome').build();

  try {

    // Step 2: Open the Google homepage
```

```
        await driver.get('https://www.google.com');

        // Step 3: Locate the search box and enter a search query
        await driver.findElement(By.name('q')).sendKeys('Selenium WebDriver', Key.RETURN);

        // Step 4: Wait for the search results to load
        await driver.wait(until.titleContains('Selenium WebDriver'), 5000);

        // Step 5: Print the page title in the console
        let title = await driver.getTitle();
        console.log('Page Title:', title);

        // Step 6: Check if the search results are displayed
        if (title.includes('Selenium WebDriver')) {
            console.log('Test Passed: Search results loaded successfully.');
        } else {
            console.log('Test Failed: Search results not loaded.');
        }
    } catch (error) {
        console.error('Error:', error);
    } finally {
        // Step 7: Close the browser
        await driver.quit();
    }
}
```

testGoogleSearch();

Explanation of the Code:

1. **Import Selenium WebDriver:**
    - **Builder: Used to create a new browser instance.**
    - **By: Used to locate elements on the web page.**
    - **Key: Simulates keyboard actions.**
    - **until: Used for waiting until a condition is met.**
2. **testGoogleSearch Function:**
    - **Step 1: Launches the Chrome browser.**
    - **Step 2: Opens https://www.google.com.**
    - **Step 3: Finds the search box using its name attribute and types a query (Selenium WebDriver).**
    - **Step 4: Waits until the page title includes "Selenium WebDriver".**
    - **Step 5: Prints the page title.**
    - **Step 6: Verifies if the search results are displayed correctly.**
    - **Step 7: Closes the browser.**

Part 4: Running the Selenium Test Script

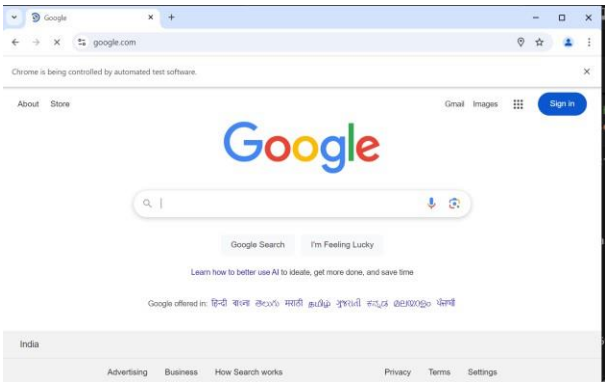
Step 1: Running the Script

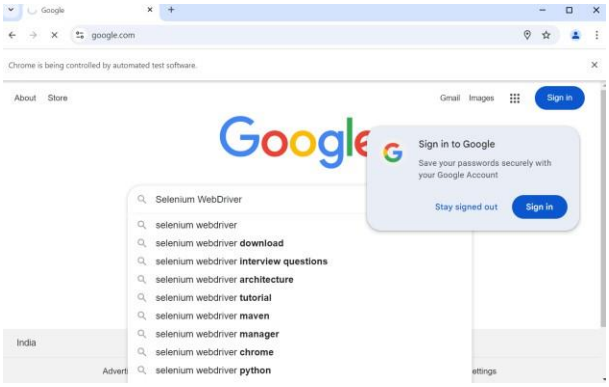
1. **Make sure that ChromeDriver is in the project folder.**
2. **Open the terminal in Visual Studio Code.**
3. **Run the test script:**

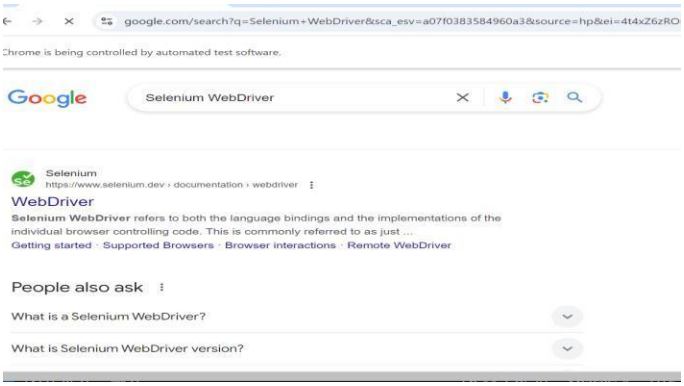
node Test.js

Expected Output:

- **The Chrome browser should open, navigate to Google, perform a search for "Selenium WebDriver", and display the search results.**
- **The terminal should display:**

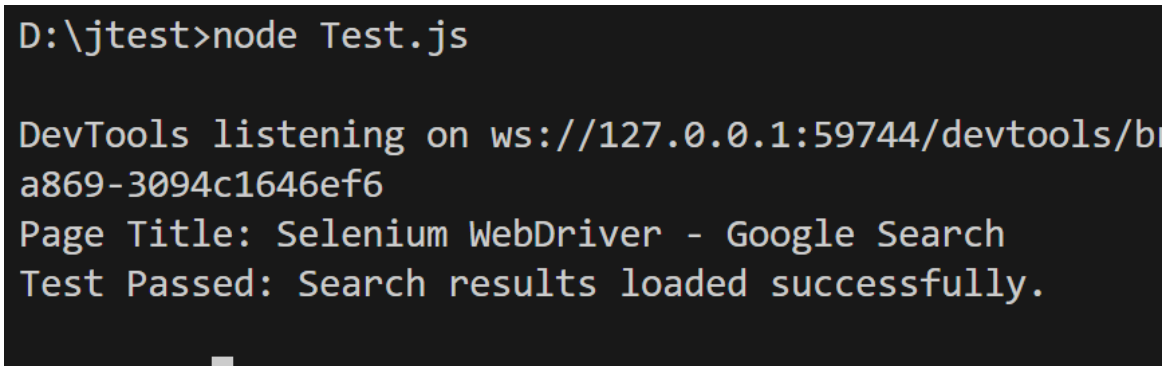






Page Title: Selenium WebDriver - Google Search

Test Passed: Search results loaded successfully.



Part 5: Exploring Selenium Commands

Commonly Used Selenium Commands

1. **Navigating to a URL:**

```
await driver.get('https://example.com');
```

2. **Locating Elements:**
   - **By ID:**

```
await driver.findElement(By.id('elementId'));
```

   - **By Class Name:**

```
await driver.findElement(By.className('className'));
```

   - **By Name:**

```
await driver.findElement(By.name('elementName'));
```

3. **Performing Actions:**
   - **Click an Element:**

```
await driver.findElement(By.id('buttonId')).click();
```

   - **Enter Text:**

```
await driver.findElement(By.name('inputName')).sendKeys('Hello World');
```

4. **Waiting for Elements:**

```
await driver.wait(until.elementLocated(By.id('elementId')), 5000);
```

5. **Closing the Browser:**

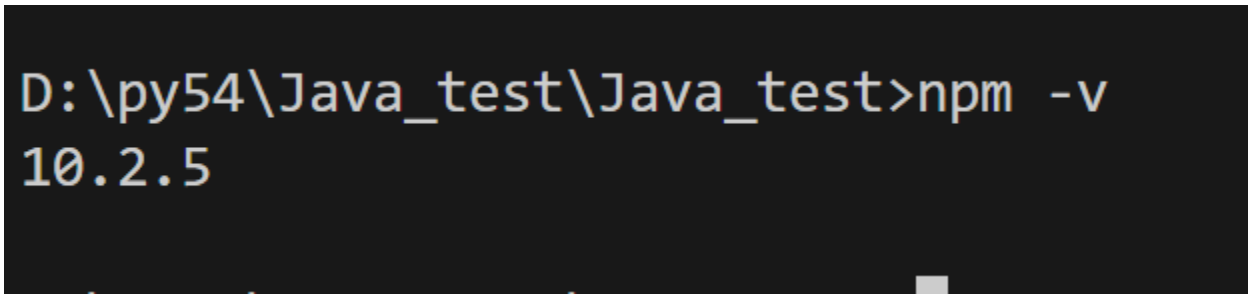
```
await driver.quit();
```

11 JavaScript Registration Form Testing using Selenium

Objective:

To create a simple registration form using HTML and JavaScript, and perform automated functional testing using Selenium.

Firstly we have to download npm and nodejs
To check whether npm is downloaded or not use this command.



To check whether nodejs is downloaded or not use this command.

nodejs –v

And then follow below steps

Step 1: Creating the Registration Form

1. Open Visual Studio Code.
2. Create a folder named RegistrationForm.
3. Inside the folder, create a file named index.html with the following content:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Registration Form</title>

<!-- Inline CSS -->

```
<style>
    body {
      font-family: Arial, sans-serif;
      text-align: center;
      margin-top: 50px;
      background-color: #f5f5f5;
    }
    form {
      display: inline-block;
      text-align: left;
      background-color: #ffffff;
      padding: 20px;
      border-radius: 8px;
      box-shadow: 0 4px 8px rgba(0, 0, 0, 0.1);
    }
    input {
      width: 100%;
      padding: 10px;
      margin: 10px 0;
      box-sizing: border-box;
      border-radius: 4px;
      border: 1px solid #ccc;
    }
    button {
      background-color: #4CAF50;
color: white;
      padding: 10px 20px;
      border: none;
      border-radius: 4px;
```

```
        cursor: pointer;
    }
button:hover {
        background-color: #45a049;
    }
    #message {
color: green;
        font-weight: bold;
        margin-top: 20px;
    }
    .error {
color: red;
    }
</style>
</head>
<body>
<h2>Registration Form</h2>
<!-- Registration Form -->
<form id="registrationForm">
<label for="username">Username:</label><br>
<input type="text" id="username" name="username" placeholder="Enter username" required><br>

<label for="email">Email:</label><br>
<input type="email" id="email" name="email" placeholder="Enter email" required><br>

<label for="password">Password:</label><br>
<input type="password" id="password" name="password" placeholder="Enter password" required><br>
```

```
<button type="submit">Register</button>
</form>

<!-- Message Display -->
<p id="message"></p>

<!-- Inline JavaScript -->
<script>
    document.getElementById("registrationForm").addEventListener("submit",
function(event) {
event.preventDefault(); // Prevent form from submitting

        // Get input values
const username = document.getElementById("username").value;
const email = document.getElementById("email").value;
const password = document.getElementById("password").value;
constmessageElement = document.getElementById("message");

        // Validate inputs
        if (username === "" || email === "" || password === "") {
messageElement.textContent = "All fields are required!";
messageElement.classList.add("error");
            return;
        }

        // Check for valid email format
constemailPattern = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
        if (!emailPattern.test(email)) {
messageElement.textContent = "Invalid email format!";
messageElement.classList.add("error");
```

```
            return;
        }

        // If all validations pass, show success message
    messageElement.textContent = "Registration Successful!";
    messageElement.classList.remove("error");
    messageElement.style.color = "green";

        // Clear the form
    document.getElementById("registrationForm").reset();
      });
    </script>
    </body>
    </html>
```

Step 2: Running the Web Application

1. Open the terminal in Visual Studio Code.
2. Use the following command to start a simple HTTP server:

```
npx http-server
```

3. Open your browser and go to http://localhost:8080 to view the form.

Part 2: Testing the Registration Form using Selenium

Step 1: Setting up Selenium in Node.js

1. Initialize a Node.js project:

```
npminit–y
```

```
D:\>mkdir 11th

D:\>cd 11th

D:\11th>npm init -y
Wrote to D:\11th\package.json:

{
  "name": "11th",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

2. Install the Selenium WebDriver package:

npm install selenium-webdriver

3. Download the ChromeDriver that matches your Chrome browser version from ChromeDriver Downloads.
4. Extract the downloaded chromedriver and place it in the project folder.

Step 2: Writing the Selenium Test Script

Create a file named test.js and add the following code:

javascript

```
const { Builder, By, Key, until } = require('selenium-webdriver');


async function testRegistrationForm() {
  // Set up WebDriver
let driver = await new Builder().forBrowser('chrome').build();
try {
    // Navigate to the registration form
awaitdriver.get('http://localhost:8080');


    // Enter username
```

```
awaitdriver.findElement(By.id('username')).sendKeys('student1');

    // Enter email
await driver.findElement(By.id('email')).sendKeys('student1@example.com');

    // Enter password
awaitdriver.findElement(By.id('password')).sendKeys('password123');

    // Click the Register button
awaitdriver.findElement(By.css('button[type="submit"]')).click();

    // Wait for the success message to appear
awaitdriver.wait(until.elementLocated(By.id('message')), 5000);

    // Get the message text
let message = await driver.findElement(By.id('message')).getText();
console.log('Test Result:', message);

    // Check if registration is successful
if (message === 'Registration Successful!') {
console.log('Test Passed');
    } else {
console.log('Test Failed');
    }
  } finally {
    // Close the browser
```

```
awaitdriver.quit();
  }
}

testRegistrationForm();
```

Step 3: Running the Selenium Test

1. Ensure your HTTP server is running:

```
npx http-server
```

2. Open a new terminal and run the Selenium test script:

```
node test.js
```

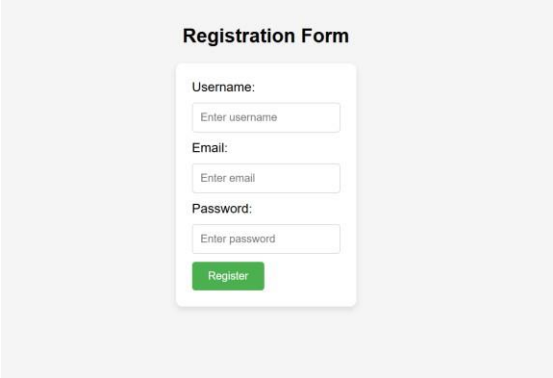
```
D:\11th>node Test.js

DevTools listening on ws://127.0.0.1:57433/devtools/browser/86a9542d-b92e-4d90-
a488-57ec6277b090
Test Result: Registration Successful!
Test Passed
```

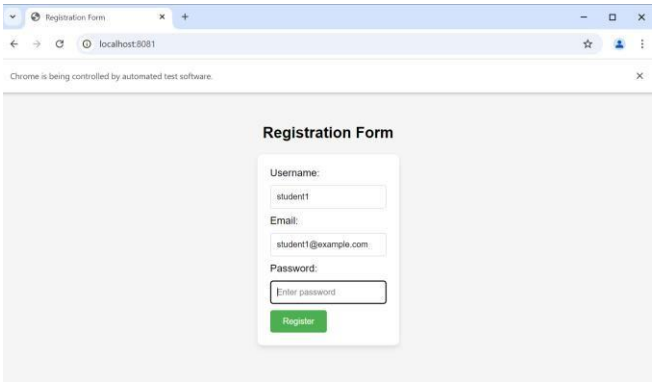
Expected Output:

- The console should display:

```
Test Result: Registration Successful!
Test Passed
```


```
D:\11th>node Test.js

DevTools listening on ws://127.0.0.1:57433/devtools/browser/86a9542d-b92e-4d90-
a488-57ec6277b090
Test Result: Registration Successful!
Test Passed
```


Registration Form
Username:
Enter username
Email:
Enter email
Password:
Enter password
Register


Registration Form
localhost:8081
Chrome is being controlled by automated test software.
Registration Form
Username:
student1
Email:
student1@example.com
Password:
Enter password
Register

12 Node.js Registration Form, Docker Containerization, and Selenium Testing

Overview:

This project consists of a simple **Node.js registration form** hosted in a **Docker container**, with automated testing using **Selenium WebDriver**. The main steps involved are:

1. Setting up a Node.js registration form.
2. Dockerizing the application.
3. Writing Selenium tests to automate the form submission and validate the result.

Step 1: Create a Node.js Registration Form

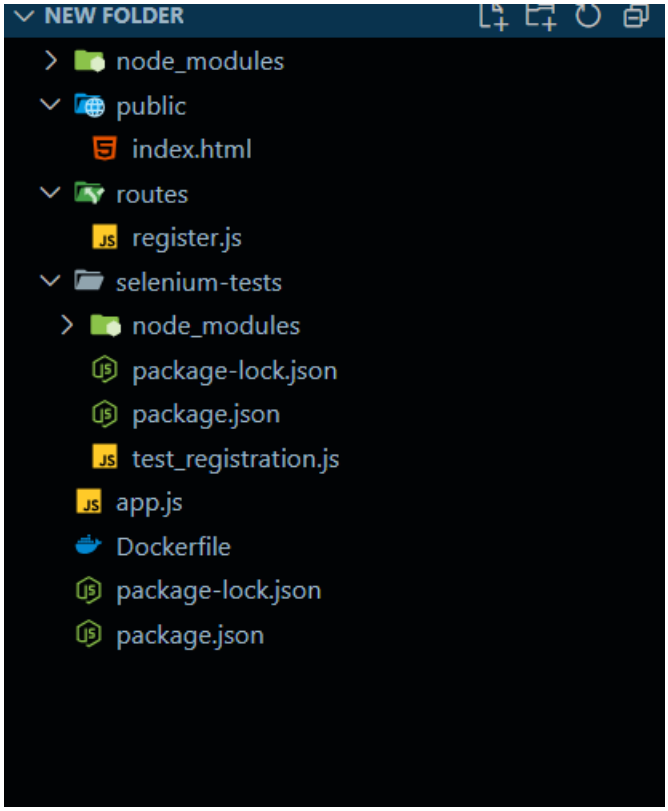
1.1 package.json

The package.json file defines the project metadata and dependencies. It includes necessary packages like express for the server and body-parser for handling form submissions.

express: A lightweight web framework for Node.js that simplifies routing and HTTP handling.

body-parser: Middleware to parse the incoming request bodies, enabling us to handle form submissions easily.

Project Structure




```
{
  "name": "nodejs-registration",
  "version": "1.0.0",
  "description": "Simple Node.js registration form",
  "main": "app.js",
  "scripts": {
    "start": "node app.js"
  },
  "dependencies": {
    "express": "^4.18.2",
    "body-parser": "^1.20.2"
  }
```

}

Create a file app.js

1.2 app.js

```
const express = require('express');

constbodyParser = require('body-parser');

const path = require('path');

const app = express();

// Middleware to handle form data

app.use(bodyParser.urlencoded({ extended: true }));

app.use(express.static('public'));


// Serve registration form

app.get('/', (req, res) => {

res.sendFile(path.join(__dirname, 'public', 'index.html'));

});

// Handle form submission

app.post('/register', (req, res) => {

const{ username, email, password } = req.body;

console.log(`Username: ${username}, Email: ${email}`);

res.send('Registration successful!');

});


// Start the server on port 3000

const PORT = 3000;
```

```
app.listen(PORT, () => {

console.log(`Server is running on http://localhost:${PORT}`);

});
```

Express server setup: Sets up an Express server that listens on port 3000.
Static files: Serves static files (like index.html) from the public/ folder.

POST route: Handles registration form submissions and logs the data to the console.

1.3 public/index.html

This is the HTML form where users can input their registration details (username, email, and password). Upon submission, the data is sent to the server.

```
<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Registration Form</title>

</head>

<body>

<h2>Registration Form</h2>

<form method="POST" action="/register">

<label for="username">Username:</label>

<input type="text" id="username" name="username" required><br>

<label for="email">Email:</label>

<input type="email" id="email" name="email" required><br>

<label for="password">Password:</label>

<input type="password" id="password" name="password" required><br>

<button type="submit">Register</button>
```

</form>

</body>

</html>

Step 2: Create a Dockerfile

A **Dockerfile** is used to containerize the Node.js application. It includes instructions to install dependencies, copy the application code, and run the app inside a container.

```
# Use Node.js base image
FROM node:14

# Set the working directory
WORKDIR /usr/src/app

# Copy package.json and install dependencies
COPYpackage.json ./
RUNnpm install

# Copy application files
COPY . .

# Expose port 3000
EXPOSE 3000

# Start the application

CMD ["npm", "start"]
```

Base image: We use the official node:14 Docker image as a base.
Dependencies: It copies the package.json file and installs the dependencies.
Expose port: Port 3000 is exposed, which is the port the application runs on.
CMD: Runs npm start to launch the app in the container

Step 3: Build and Run Docker Container

To build and run the Docker container, execute the following commands:

Build the Docker image

docker build -t nodejs-registration-app .

```
PS C:\Users\shiva kumar\OneDrive\Desktop\New folder> docker build -t nodejs-registration-app .
[+] Building 3.6s (11/11) FINISHED
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 341B
 => [internal] load metadata for docker.io/library/node:14
 => [auth] library/node:pull token for registry-1.docker.io
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [internal] load build context
 => => transferring context: 209.40kB
 => [1/5] FROM docker.io/library/node:14@sha256:a158d3b9b4e3fa813fa6c8c590b8f0a860e015ad4e59bbce5744d2f6fd8461aa
 => CACHED [2/5] WORKDIR /usr/src/app
 => CACHED [3/5] COPY package.json ./
 => CACHED [4/5] RUN npm install
 => CACHED [5/5] COPY . .
 => exporting to image
 => => exporting layers
 => => writing image sha256:8c9e1a9f520369ce54eb60ed83ceb3ab077dc1c28349781ca88484acfe4cd6aa
 => => naming to docker.io/library/nodejs-registration-app

What's Next?
```

Run the container

docker run -d -p 3000:3000 --name nodejs-registration nodejs-registration-app

```
What's Next?
  View a summary of image vulnerabilities and recommendations → docker scout quickview
PS C:\Users\shiva kumar\OneDrive\Desktop\New folder> docker run -d -p 3000:3000 --name my-nodejs-app nodejs-registration-app
66c50e1726fe3d7a878564677c3aad33fc6050aa2667e4c20d83a6c35dde6a27
PS C:\Users\shiva kumar\OneDrive\Desktop\New folder>
```

After running the container, the app is accessible at http://localhost:3000.

Step 4: Set Up Selenium for Automated Testing

To automate testing for the registration form, we use **Selenium WebDriver** to simulate a user filling out the form and submitting it.

4.1 Install Selenium WebDriver

Inside the `selenium-tests` directory, initialize a new Node.js project and install `selenium-webdriver` and `chromedriver`:

By following command

Npm init -y

npm install selenium-webdriverchromedriver

4.2 Create test_registration.js

This file contains the test logic for Selenium, which fills out the registration form and checks if the registration is successful.

Copy the command

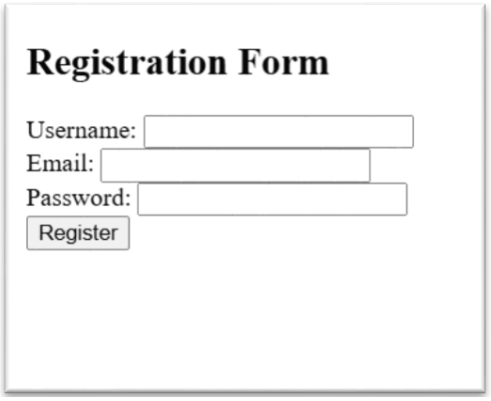
```
const{ Builder, By, until } = require('selenium-webdriver');
const chrome = require('selenium-webdriver/chrome');

async function runTest() {
    let driver = await new Builder().forBrowser('chrome').build();

    try {
        // Open the registration form
        await driver.get('http://localhost:3000');

        // Fill in the registration form
        await driver.findElement(By.id('username')).sendKeys('testuser');
        await driver.findElement(By.id('email')).sendKeys('testuser@example.com');
        await driver.findElement(By.id('password')).sendKeys('password123');

        // Submit the form
        await driver.findElement(By.tagName('button')).click();

        // Wait for the response
        await driver.wait(until.elementLocated(By.tagName('body')), 5000);

        // Verify the result
        let bodyText = await driver.findElement(By.tagName('body')).getText();
```

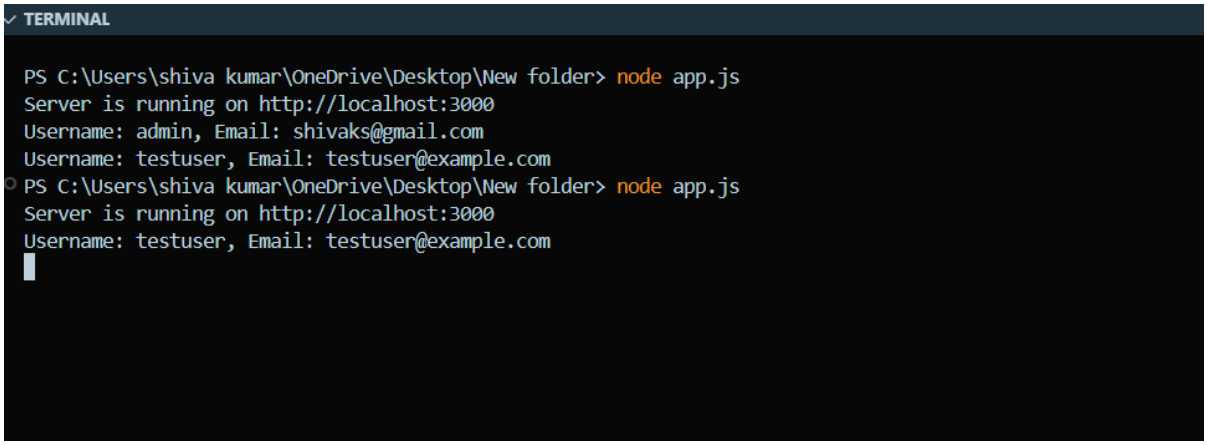
```
        if (bodyText.includes('Registration successful!')) {
console.log('Test Passed: Registration was successful.');
        } else {
console.log('Test Failed: Registration was not successful.');
        }
    } finally {
        await driver.quit();
    }
}

runTest();
```

Step 5: Run Selenium Test Cases

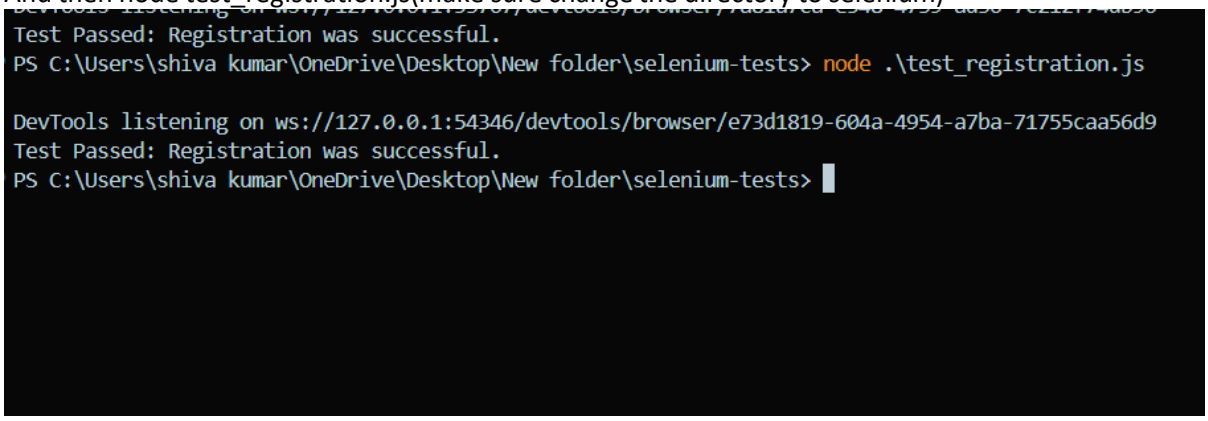
To run the automated tests, execute the following command:

First run this app.js





And then node test_registration.js(make sure change the directory to selenium)



If the registration is successful, you will see the output:

Test Passed: Registration was successful