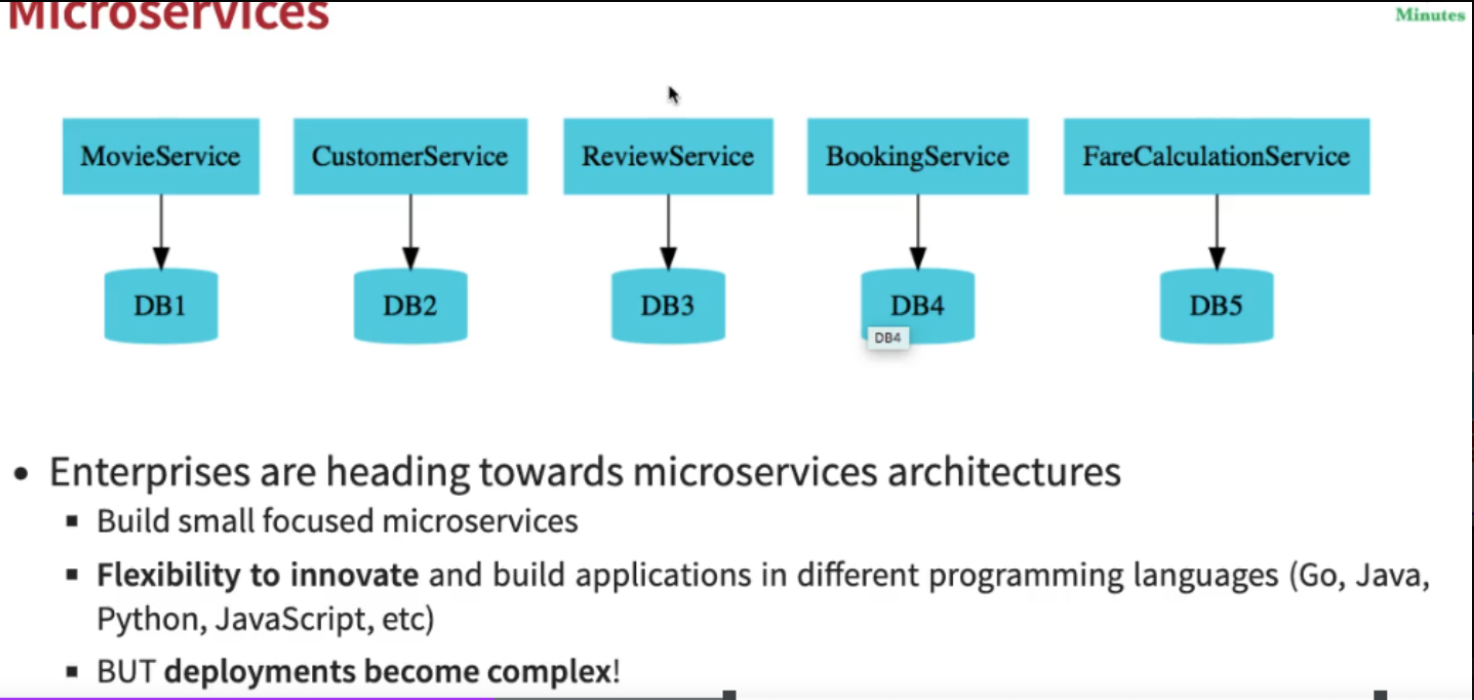
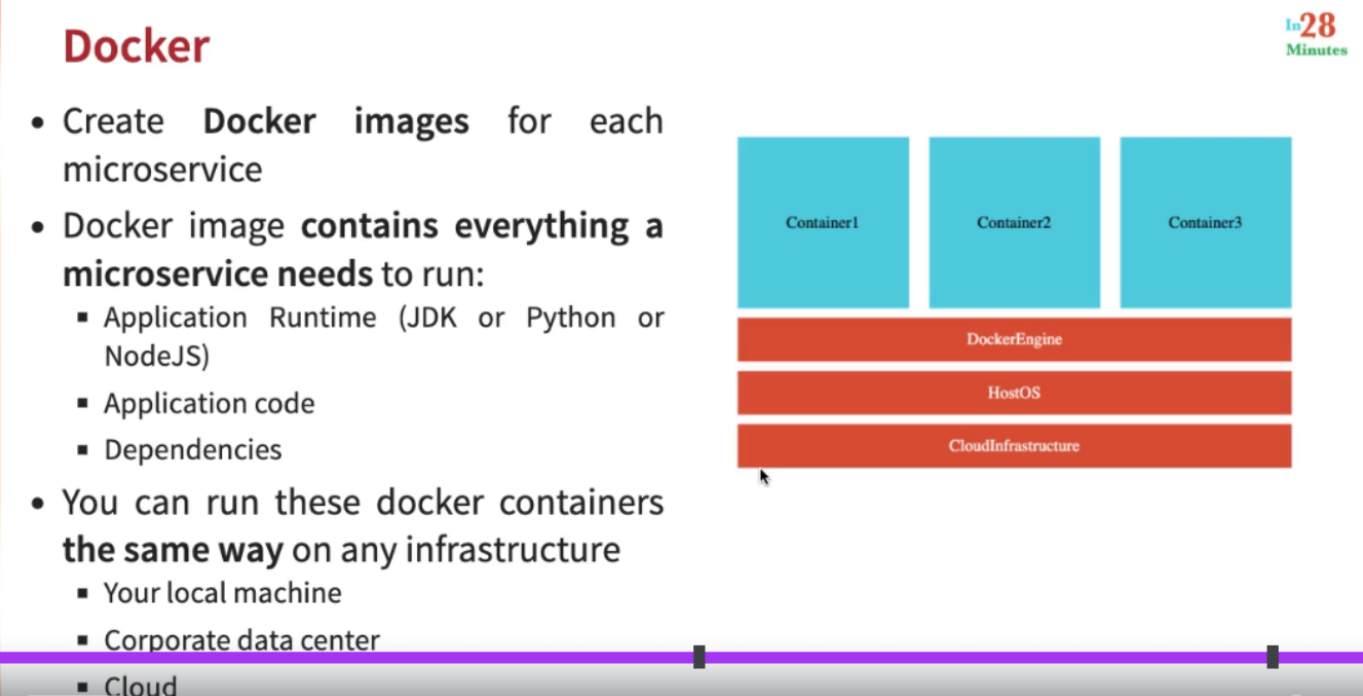
**Docker With Microservices using Spring boot and Spring Cloud**

**https://github.com/in28minutes/spring-microservices-v3**

**1.What is Docker?**

Docker is a platform that allows developers to package applications along with all their dependencies into a standardized unit called a **container**. This ensures that the application can run consistently across different environments, whether it's your local machine, a corporate data center, or the cloud.





**Why is Docker Needed?**

Looking at the **Microservices Diagram**:

* Each microservice (e.g., MovieService, CustomerService, etc.) is independent and often has its own database (DB1, DB2, etc.).
* These services might be developed in different programming languages (Java, Python, Node.js, etc.) and require specific runtime environments and dependencies.
* Managing and deploying these services individually can become complex.

With Docker:

1. **Simplified Deployment:** Each microservice can be containerized, meaning the runtime, application code, and dependencies are all bundled into a single unit.
2. **Environment Consistency:** Docker containers ensure that the service behaves the same way, whether it runs on your local machine or in production.
3. **Flexibility to Innovate:** Developers can use different languages and frameworks for each microservice without worrying about compatibility issues during deployment.

Looking at the **Docker Diagram**:

1. **Docker Containers:** Each microservice (e.g., Container1 for MovieService) runs in its own isolated container.
2. **Docker Engine:** The Docker engine acts as the runtime for these containers, managing their lifecycle and ensuring they are lightweight and fast.
3. **Host OS:** Docker uses the underlying Host Operating System (OS), reducing overhead compared to traditional virtual machines.
4. **Cloud Infrastructure:** Containers can easily be deployed on cloud environments like AWS, Azure, or Google Cloud, making scaling and managing applications easier.

**Benefits of Docker**

1. **Isolation:** Each microservice runs in its own container, avoiding conflicts between services.
2. **Portability:** Containers can run on any machine with Docker installed, ensuring consistent behavior.
3. **Efficiency:** Docker containers are lightweight and start up quickly compared to virtual machines.
4. **Scalability:** Services can be scaled horizontally by running multiple containers.

Looking at the **Docker Diagram**:

1. **Docker Containers:** Each microservice (e.g., Container1 for MovieService) runs in its own isolated container.
2. **Docker Engine:** The Docker engine acts as the runtime for these containers, managing their lifecycle and ensuring they are lightweight and fast.
3. **Host OS:** Docker uses the underlying Host Operating System (OS), reducing overhead compared to traditional virtual machines.
4. **Cloud Infrastructure:** Containers can easily be deployed on cloud environments like AWS, Azure, or Google Cloud, making scaling and managing applications easier.

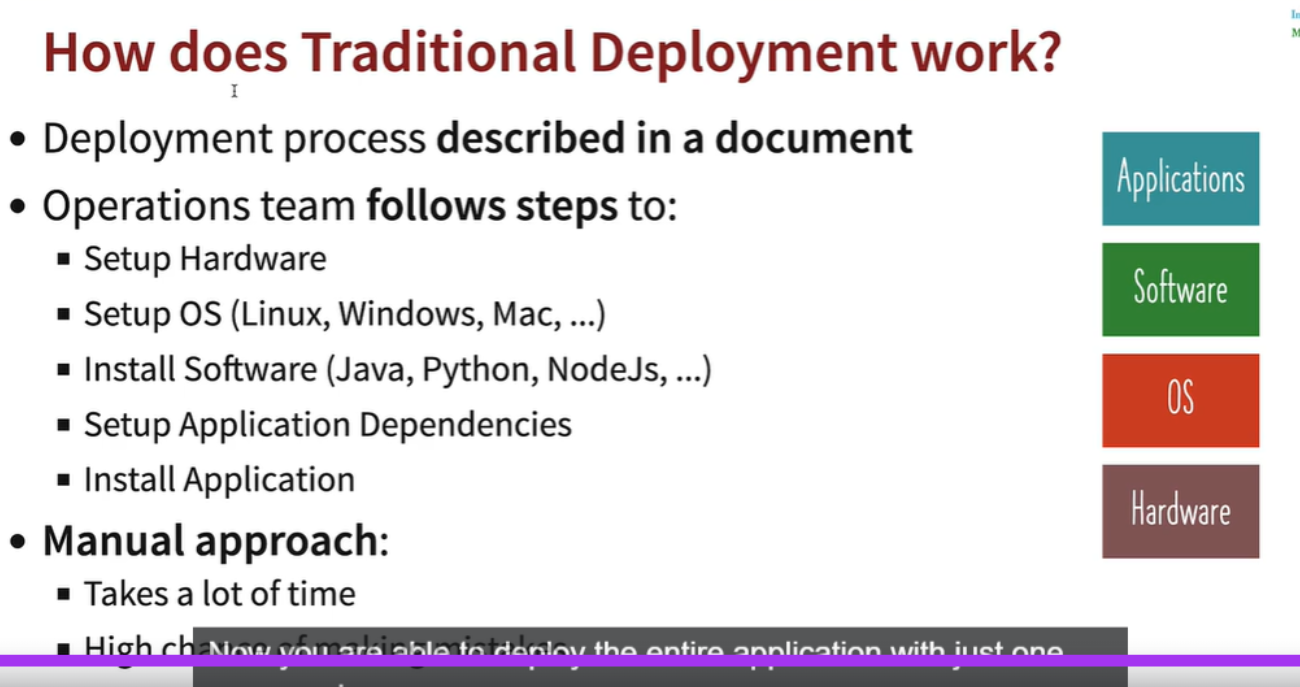
**Benefits of Docker**

1. **Isolation:** Each microservice runs in its own container, avoiding conflicts between services.
2. **Portability:** Containers can run on any machine with Docker installed, ensuring consistent behavior.
3. **Efficiency:** Docker containers are lightweight and start up quickly compared to virtual machines.
4. **Scalability:** Services can be scaled horizontally by running multiple containers.

In summary, Docker simplifies the complex deployment and management of microservices by packaging them into isolated containers, ensuring consistency, efficiency, and portability across different environments.

1. docker container run -d -p 5000:5000 in28min/hello-world-nodejs:0.0.1.RELEASE
2. docker container run -d -p 5000:5000 in28min/hello-world-java:0.0.1.RELEASE
3. docker container run -d -p 5000:5000 in28min/hello-world-python:0.0.1.RELEASE
4. docker container ls
5. docker image ls
6. docker container stop cc
7. docker container run -d -p 5001:5000 in28min/hello-world-nodejs:0.0.1.RELEASE
8. docker container run -d -p 5002:5000 in28min/hello-world-nodejs:0.0.1.RELEASE
9. docker container run -p 5003:5000 in28min/hello-world-nodejs:0.0.1.RELEASE
10. docker container run -p 5003:5000 in28min/hello-world-nodejs:0.0.1.RELEASE
12. docker --version
13. docker container ls
14. docker build -t in28min/hello-world-docker:v1 .
15. docker image list
16. docker run -d -p 5000:5000 in28min/hello-world-docker:v1
17. docker build -t in28min/hello-world-docker:v2 .
18. docker container run -d -p 5000:5000 in28min/hello-world-docker:v2
19. docker build -t in28min/hello-world-docker:v3 .
20. docker container run -d -p 5000:5000 in28min/hello-world-docker:v3
21. docker build -t in28min/hello-world-docker:v4 .

**2.Understanding Docker Fundamentals**



This image explains **how traditional deployment works** and highlights its challenges:

**Key Points in Traditional Deployment:**

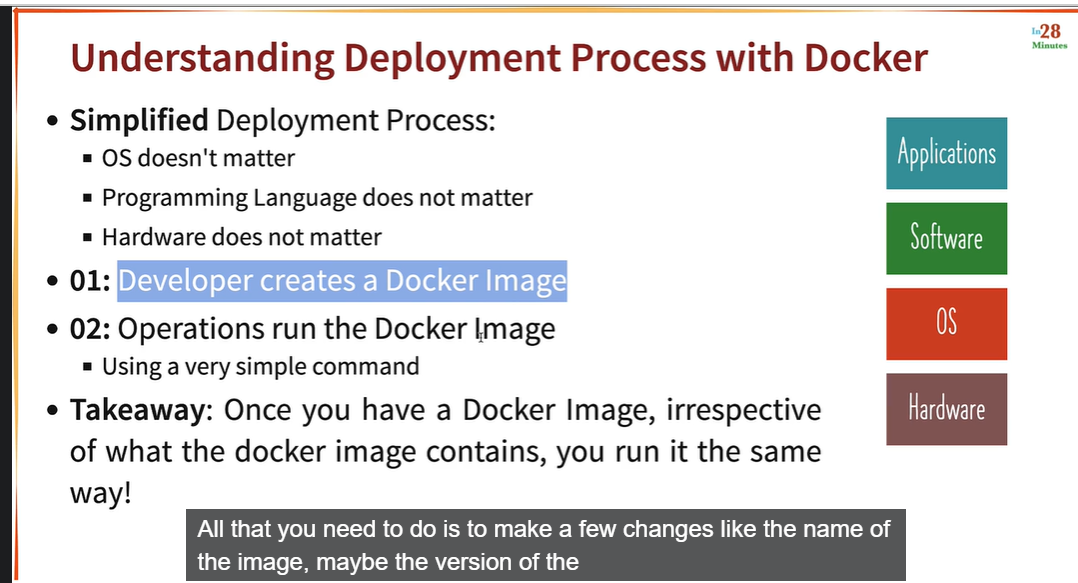
1. **Deployment Process:**
   * The process is described in a document detailing the steps needed for deployment.
   * The operations team manually follows these steps.
2. **Steps Involved:**
   * **Setup Hardware:** Provision physical or virtual machines.
   * **Setup OS:** Install and configure the required operating system (Linux, Windows, etc.).
   * **Install Software:** Install dependencies like Java, Python, Node.js, etc., required for the application to run.
   * **Setup Application Dependencies:** Configure application-specific libraries, frameworks, or services.
   * **Install Application:** Deploy the application manually by following the outlined process.
3. **Challenges of the Manual Approach:**
   * **Time-Consuming:** Each step takes time, and delays are common when scaling or debugging.
   * **Error-Prone:** High chances of manual errors due to misconfigurations or missed steps.
   * **Lack of Consistency:** Deployments may vary across environments due to human involvement, leading to issues in production.

**Why Modern Solutions Like Docker are Needed:**

The manual approach requires significant effort and is not scalable. Docker and containerization can address these problems by:

* Bundling applications with their runtime, libraries, and dependencies in **Docker images**.
* Ensuring consistency across environments.
* Allowing faster and repeatable deployments with minimal manual intervention.

This makes deployment faster, automated, and more reliable, unlike traditional methods.



This image illustrates how **Docker simplifies the deployment process** and why it is highly beneficial in modern software development, particularly for microservices architecture. Here's a breakdown of the explanation:

**1. Simplified Deployment Process**

* **OS Doesn't Matter:** With Docker, the operating system used in development, testing, and production doesn't need to match. Docker abstracts the environment differences.
* **Programming Language Doesn't Matter:** Developers can work with any programming language (e.g., Java, Python, Node.js). Docker ensures consistency across environments regardless of the language.
* **Hardware Doesn't Matter:** Whether you're running on a local machine, a corporate data center, or cloud infrastructure, Docker ensures the same behavior everywhere.

**2. Key Steps**

* **Step 01: Developer Creates a Docker Image**
  + A Docker image is essentially a blueprint of an application. It includes:
    - Application code.
    - Application runtime (e.g., Java Runtime Environment, Python interpreter).
    - Dependencies (e.g., libraries, frameworks).
    - System-level dependencies (e.g., operating system packages).
  + Once the image is created, it is shared with the operations team or deployed to a Docker registry (e.g., Docker Hub).
* **Step 02: Operations Run the Docker Image**
  + The operations team runs the Docker image using a simple command

docker run <image-name>

* + he same image can run on any system with Docker installed, ensuring the application behaves identically everywhere.

**3. Takeaway**

* Once a Docker image is created, it doesn't matter:
  + **Where it's run:** On a developer's local machine, in staging, or in production.
  + **What environment it runs on:** Local, data center, or cloud infrastructure.
* **Uniformity:** The image ensures consistency. The application behaves the same way across all platforms.

**How It Solves Traditional Deployment Challenges**

* Traditional deployment relies on step-by-step setup (hardware, OS, software, dependencies), which is time-consuming and error-prone.
* Docker eliminates these manual steps, as the entire application and its dependencies are packaged in a single image.

In summary, Docker revolutionizes the deployment process by making it simple, consistent, and efficient. It ensures that developers and operations teams spend less time on environment setup and focus more on building and delivering applications.

When you run the command:

bash

Copy code

docker container run -d -p 5000:5000 in28min/hello-world-java:0.0.1.RELEASE

here’s how Docker processes this command internally step-by-step:

**1. Pull the Docker Image**

* **What happens:** Docker first checks if the image in28min/hello-world-java:0.0.1.RELEASE exists locally on your system.
  + If the image is not found locally, Docker pulls it from the **Docker Hub** (or another configured Docker registry).
  + If the image is already present locally, Docker skips this step.

**2. Create a New Container**

* **What happens:** Docker creates a new container based on the specified image in28min/hello-world-java:0.0.1.RELEASE.
  + A **container** is an isolated runtime environment created from a Docker image.
  + The container is given a unique ID and metadata, which includes details like the image name, ports, and environment variables.

**3. Network Setup**

* **What happens:** Docker sets up the networking for the container.
  + In this case, the -p 5000:5000 flag maps:
    - **Host port 5000** to **container port 5000**.
    - Any incoming requests to http://localhost:5000 on your machine are routed to port 5000 inside the container.
  + Docker creates a virtual network bridge (default bridge network) to handle communication between the host and the container.

**4. Start the Container**

* **What happens:** The container is started in the background (-d flag means **detached mode**).
  + Docker runs the application or process defined in the image's CMD or ENTRYPOINT directive (specified in the Dockerfile for in28min/hello-world-java).
  + The container’s main process runs in isolation.

**5. Expose the Container's Port**

* **What happens:** Docker ensures that the container's exposed port (5000 in this case) is mapped to the host machine's port (5000), allowing external access.
  + Any traffic to localhost:5000 is now forwarded to the container's process running on its port 5000.

**6. Detached Mode**

* **What happens:** Since the -d flag is used, the container runs in the background. The terminal shows only the **container ID**, not the container’s logs or output.
  + You can view the logs later using:

bash

Copy code

docker logs <container-id>

**7. Resource Management**

* **What happens:** Docker isolates the container’s resources (e.g., CPU, memory, filesystem) from the host system using namespaces and control groups (cgroups).
  + The container is lightweight, running only the application and its dependencies, not a full OS.

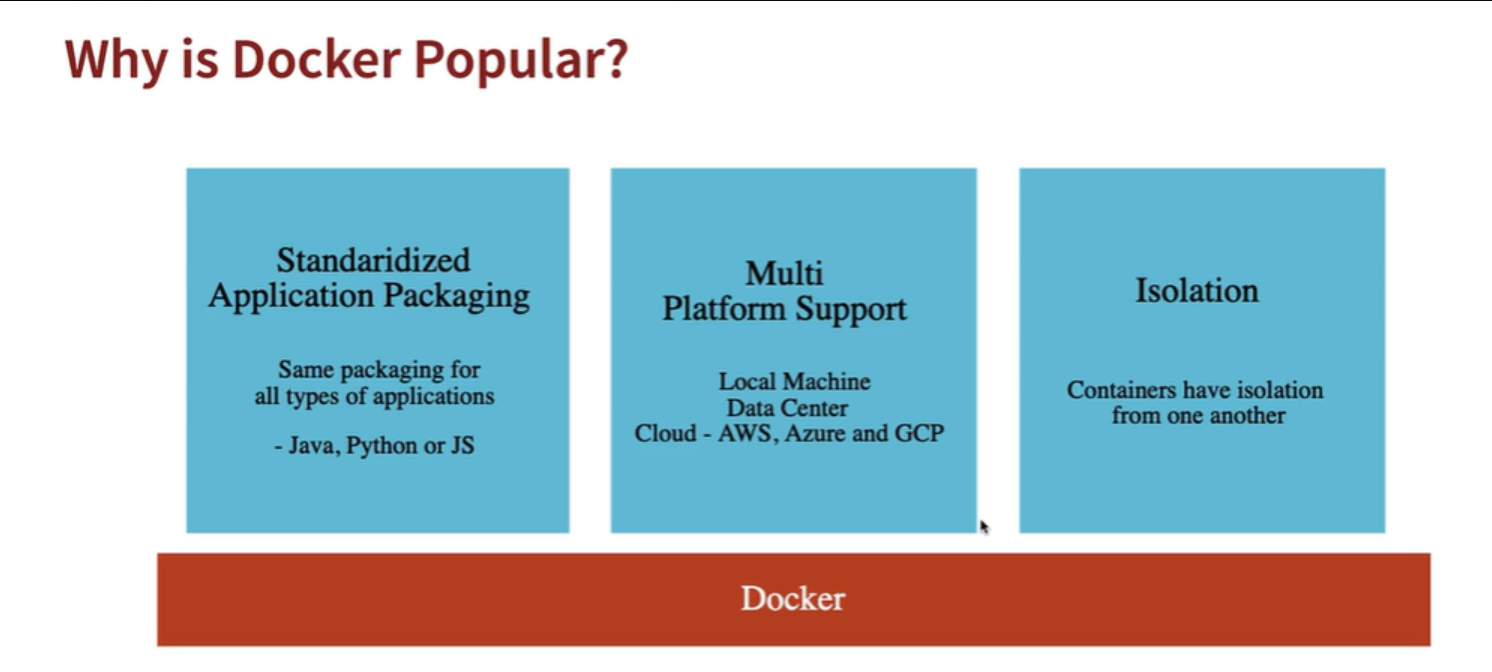
**How This All Ties Together**

1. **Image → Container:** A Docker image is a template; the docker container run command creates a running instance of it (a container).
2. **Port Binding:** The -p 5000:5000 flag allows external requests to interact with the container through your machine's port 5000.
3. **Detached Mode:** The -d flag runs the container in the background, freeing your terminal for other tasks.

**End Result**

* After the command completes, the application defined in the in28min/hello-world-java:0.0.1.RELEASE image is running in an isolated container.
* You can access it at http://localhost:5000 on your host machine. Docker takes care of routing traffic and isolating the environment for seamless execution.

**3.Understanding How Docker Works**

****

This image explains why **Docker is popular** in modern software development and deployment. Let’s break down the key points:

**1. Standardized Application Packaging**

* **What it means:**
  + Docker provides a consistent way to package applications, regardless of the programming language or framework (e.g., Java, Python, or JavaScript).
  + Developers can use a **Dockerfile** to define how their application and its dependencies are packaged into an image.
* **Why it’s beneficial:**
  + Removes inconsistencies between development, testing, and production environments.
  + Ensures that an application runs the same way across all stages of the lifecycle.
  + Simplifies the deployment process by using a single standard.

**2. Multi-Platform Support**

* **What it means:**
  + Docker containers can run on various platforms without modification. These include:
    - **Local Machine**: Your laptop or desktop.
    - **Data Center**: On-premise servers or virtual machines.
    - **Cloud**: Major cloud providers like AWS, Azure, and GCP support Docker.
* **Why it’s beneficial:**
  + Provides flexibility to deploy applications anywhere.
  + Eliminates platform-specific compatibility issues.
  + Enables seamless migration from local development to cloud deployment.

**3. Isolation**

* **What it means:**
  + Docker containers are isolated environments:
    - Each container has its own file system, processes, memory, and network stack.
    - Containers run independently of each other, even on the same host machine.
* **Why it’s beneficial:**
  + Ensures that the behavior of one application does not affect others running on the same system.
  + Improves security by isolating applications and their dependencies.
  + Allows developers to run multiple versions of the same application or library side-by-side without conflict.

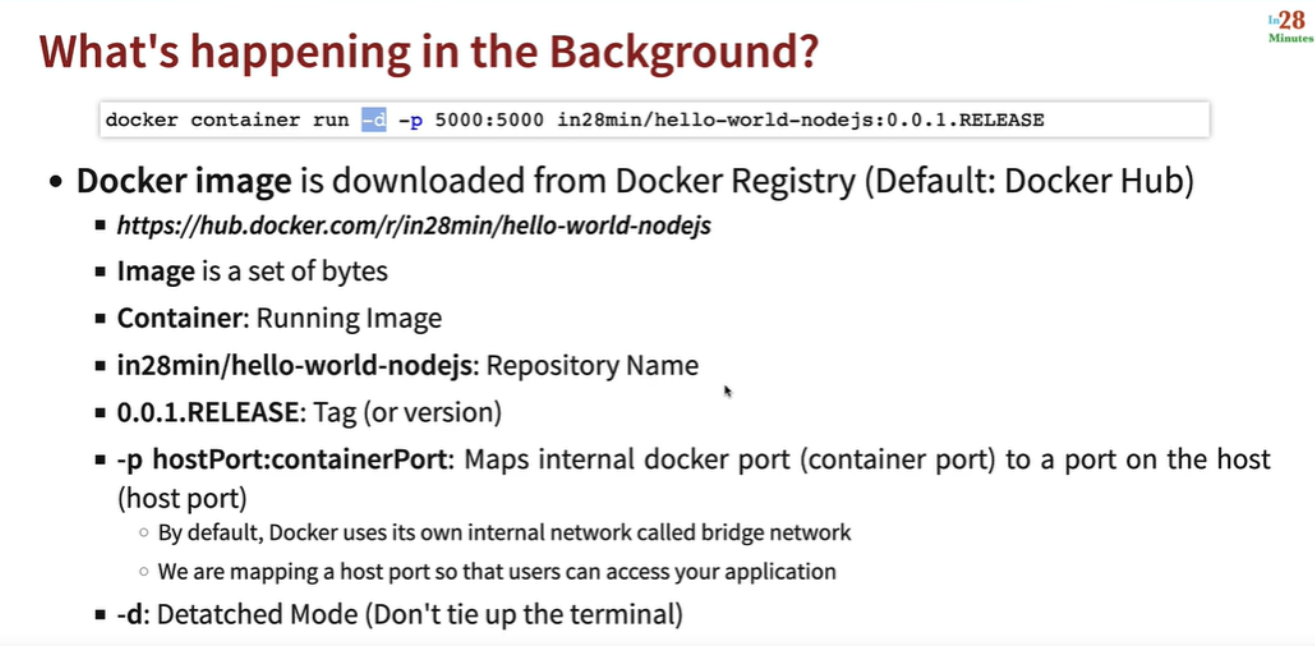
**Docker’s Value**

* Docker combines these three features to provide a powerful solution for:
  + **Simplifying deployment**: With standardized images.
  + **Enhancing portability**: Across local, on-premise, and cloud environments.
  + **Improving reliability**: Through application isolation.

**Conclusion**

Docker is popular because it streamlines the development-to-deployment workflow. It solves challenges like inconsistent environments, platform dependencies, and application conflicts, making it a go-to choice for modern microservices and cloud-native applications.

**4. Understanding How Docker Works**

****

This image explains the steps and process happening in the background when you run the Docker command:

docker container run -d -p 5000:5000 in28min/hello-world-nodejs:0.0.1.RELEASE

**Detailed Breakdown:**

1. **Docker Image**
   * The Docker image is a blueprint containing everything needed to run the application, such as:
     + Application code.
     + Dependencies.
     + Runtime environment (like NodeJS, Java, etc.).
   * **Source**: The image is pulled from a Docker registry, typically the **Docker Hub** (default registry).
     + Example: https://hub.docker.com/r/in28min/hello-world-nodejs.
2. **Image vs. Container**
   * **Image**: A static, packaged version of the application (a set of bytes).
   * **Container**: A running instance of the image. When you execute the Docker run command, a container is created from the image.
3. **Repository Name**
   * **in28min/hello-world-nodejs**: Refers to the repository name on Docker Hub.
     + **in28min**: The user or organization owning the repository.
     + **hello-world-nodejs**: The name of the specific application.
4. **Tag or Version**
   * **0.0.1.RELEASE**: Refers to the version of the Docker image.
     + It ensures you’re running a specific version of the image.
     + Tags like latest can be used to refer to the most recent version.
5. **Port Mapping**
   * **-p 5000:5000**:
     + The first 5000: The **host port** (your machine's port).
     + The second 5000: The **container port** (the internal port of the application running inside the container).
     + This mapping ensures that any requests made to localhost:5000 on your machine are routed to the container’s application running on port 5000.
   * **Why this is needed:**
     + By default, containers use Docker’s internal network (bridge network).
     + Port mapping makes the container’s application accessible from the host machine or other networks.
6. **Detached Mode**
   * **-d**: Runs the container in detached mode, meaning:
     + The container runs in the background.
     + The terminal is not tied up, allowing you to run additional commands.

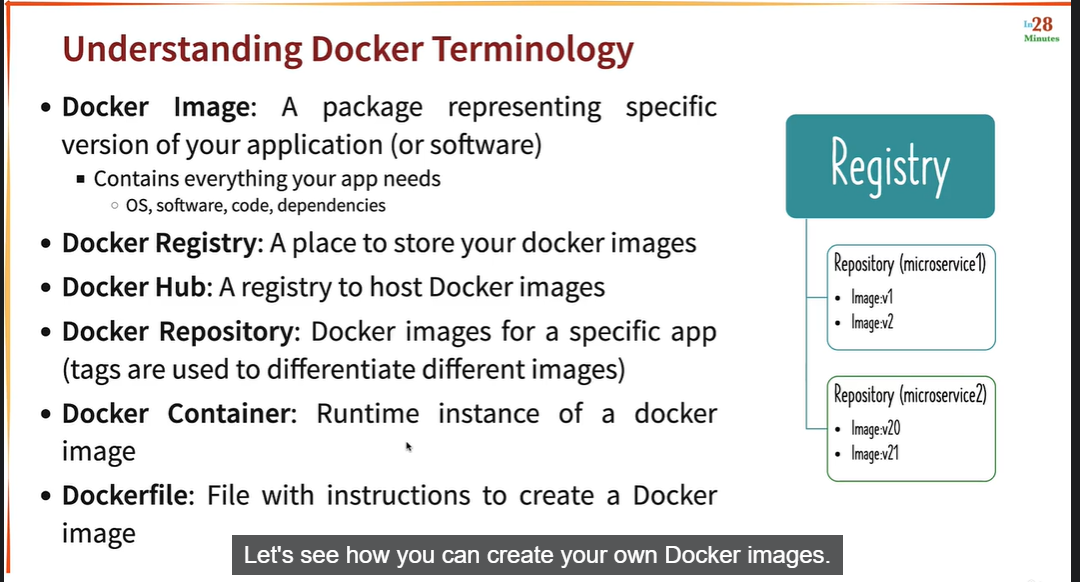
**What Happens Internally**

* The Docker engine:
  1. Pulls the image from Docker Hub (if not already available locally).
  2. Creates a container from the image.
  3. Maps the host port to the container port.
  4. Runs the container in the background.

**Takeaways**

* This command sets up and runs an application (like a Node.js app) quickly without worrying about setting up the runtime or dependencies manually.
* You can interact with the application on localhost:5000 from your host machine.

**16.Understanding Docker Terminology**



his image provides a summary of key **Docker terminology**. Let’s break it down:

**1. Docker Image**

* A **package** representing a specific version of your application or software.
* Contains:
  + **OS components** (minimal, to run your app).
  + Application **dependencies**.
  + Your **application code**.
* Think of it as a **blueprint** for your application.

**Example**: A Java application packaged with a JDK, libraries, and code.

**2. Docker Registry**

* A **storage location** for Docker images.
* **Purpose**: To store and share Docker images.
* Docker **pulls images** from a registry to run on a local machine.

**3. Docker Hub**

* A **public Docker registry** maintained by Docker Inc.
* Hosts **prebuilt images** that you can use or share with others.
* Acts as the default registry for docker pull and docker push commands.

**Example**: You can pull images like nginx, mysql, or your custom image (e.g., in28min/hello-world).

**4. Docker Repository**

* A logical grouping of **Docker images** for a specific app.
* Example: Repository **microservice1** could have multiple tagged versions:
  + **v1** (version 1 of the image).
  + **v2** (updated version of the image).

**Key Point**: Tags (e.g., v1, latest) differentiate between images.

**5. Docker Container**

* A **runtime instance** of a Docker image.
* When you run a Docker image, it creates a **container**.
* **Key Features**:
  + Isolated environment (has its own file system, network, and process space).
  + Lightweight and efficient.

**Analogy**: If the image is a **blueprint**, the container is the **building** created from that blueprint.

**6. Dockerfile**

* A **script file** containing instructions to build a Docker image.
* Includes:
  + The **base image** (e.g., FROM openjdk:8).
  + Commands to copy code, install dependencies, and set the entry point

**Example of a Dockerfile**:

FROM openjdk:8

COPY target/app.jar app.jar

ENTRYPOINT ["java", "-jar", "app.jar"]

**Visualization in the Image**

* **Registry**: A registry is a storage place for repositories.
* **Repositories**: Each microservice has its own repository with different image versions.
  + E.g., **microservice1** has images tagged as v1 and v2.
  + E.g., **microservice2** has images tagged as v20 and v21.

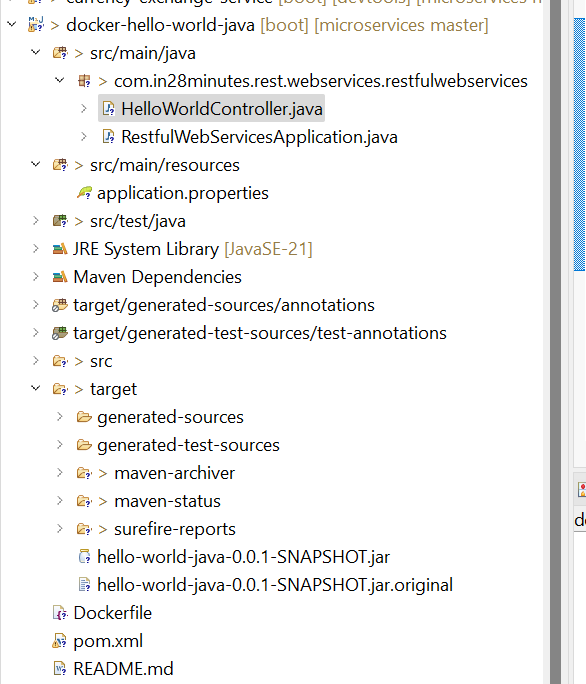
**Takeaway**

* **Images** are the building blocks (blueprints).
* **Containers** are the running instances (buildings).
* **Registries** store the images for easy reuse.
* **Dockerfiles** help automate the creation of images.

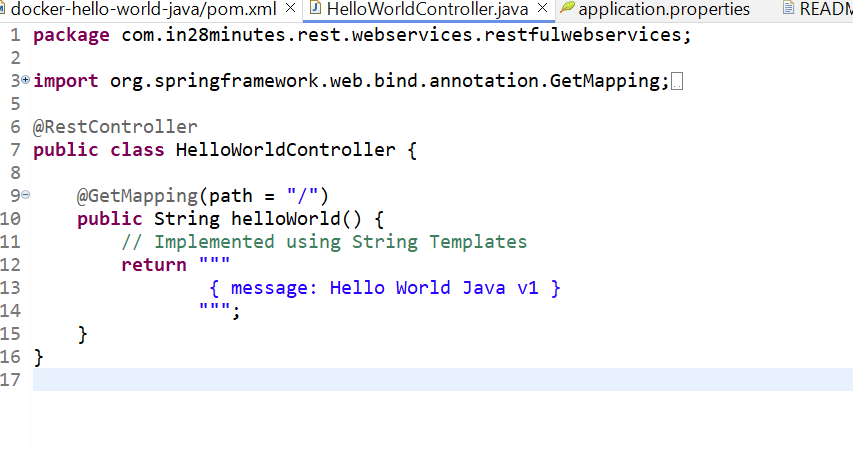
This terminology is critical to understanding how Docker streamlines application deployment and management.

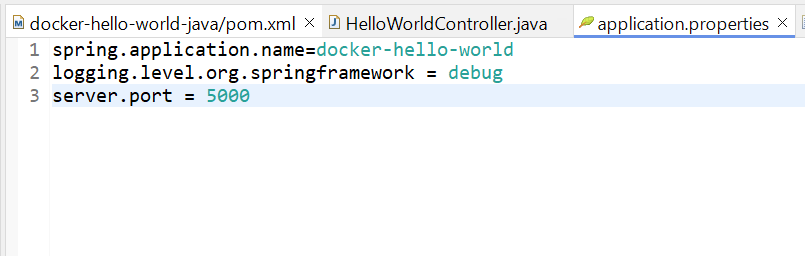
**6.Creating Docker Image for a Spring Boot Project – Dockerfile**

**Step 1 :Create one project docker-hello-world-java**

****

**Step 2 :Create one controller**

****

**Step 3 Below is the application.properties**  
  


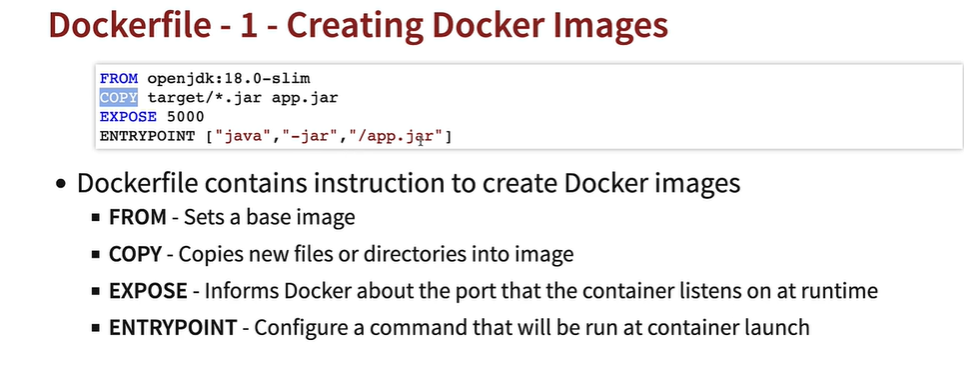
Step 4 :Create docker file in the project   
  
## Dockerfile - 1 - Creating Docker Images

FROM openjdk:21-slim

COPY target/\*.jar app.jar

EXPOSE 5000

ENTRYPOINT ["java", "-jar", "/app.jar"]

****

**Components in the Dockerfile**

1. **FROM**
   * Sets the **base image** for the Docker image.
   * Example:

dockerfile

Copy code

FROM openjdk:18.0-slim

* + This uses the openjdk:18.0-slim base image, which is a minimal version of Java OpenJDK.

1. **COPY**
   * Copies files or directories from the local system into the Docker image.
   * Example:

dockerfile

Copy code

COPY target/\*.jar app.jar

* + This command copies all .jar files from the target directory (on the host machine) into the image as app.jar.

1. **EXPOSE**
   * Specifies the **port** on which the container will listen for incoming connections.
   * Example:

dockerfile

Copy code

EXPOSE 5000

* + This tells Docker that the application inside the container will use port 5000.

1. **ENTRYPOINT**
   * Defines the **command to execute** when the container starts.
   * Example:

dockerfile

Copy code

ENTRYPOINT ["java", "-jar", "/app.jar"]

* + This runs the Java application (app.jar) using the java -jar command.

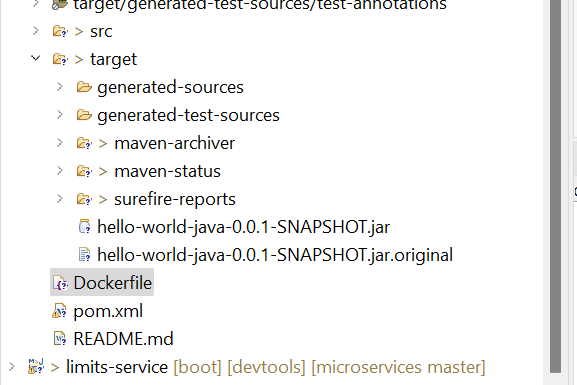
**How It Works**

1. **Base Image**:
   * The FROM instruction sets up a minimal environment with Java 18 (OpenJDK). It ensures all necessary Java libraries and runtime are available.
2. **Copy Files**:
   * The COPY command moves the compiled application JAR file (app.jar) into the Docker image.
3. **Expose Port**:
   * The EXPOSE instruction tells Docker that the application will listen for HTTP requests on port 5000.
4. **Run Command**:
   * The ENTRYPOINT specifies that the app.jar file should be executed when the container starts.

**Key Takeaways**

* **Dockerfile is used to automate image creation.**
* The image will include everything the application needs to run: Java runtime, compiled code (app.jar), and the configured port.
* When the container is started, the ENTRYPOINT ensures the application runs automatically.

By following this Dockerfile, you can create a portable and reproducible environment for your Java application.

Step 5 :  
  
Run Mvn clean install and check where .jar file has been created or not   
  


If .jar file ois created open pwershelll and go to the this directory

C:\microservices\docker-hello-world-java

Step 6: run below command

PS C:\microservices\docker-hello-world-java> docker –version

Response : Docker version 27.4.0, build bde2b89

Step 7 run below command

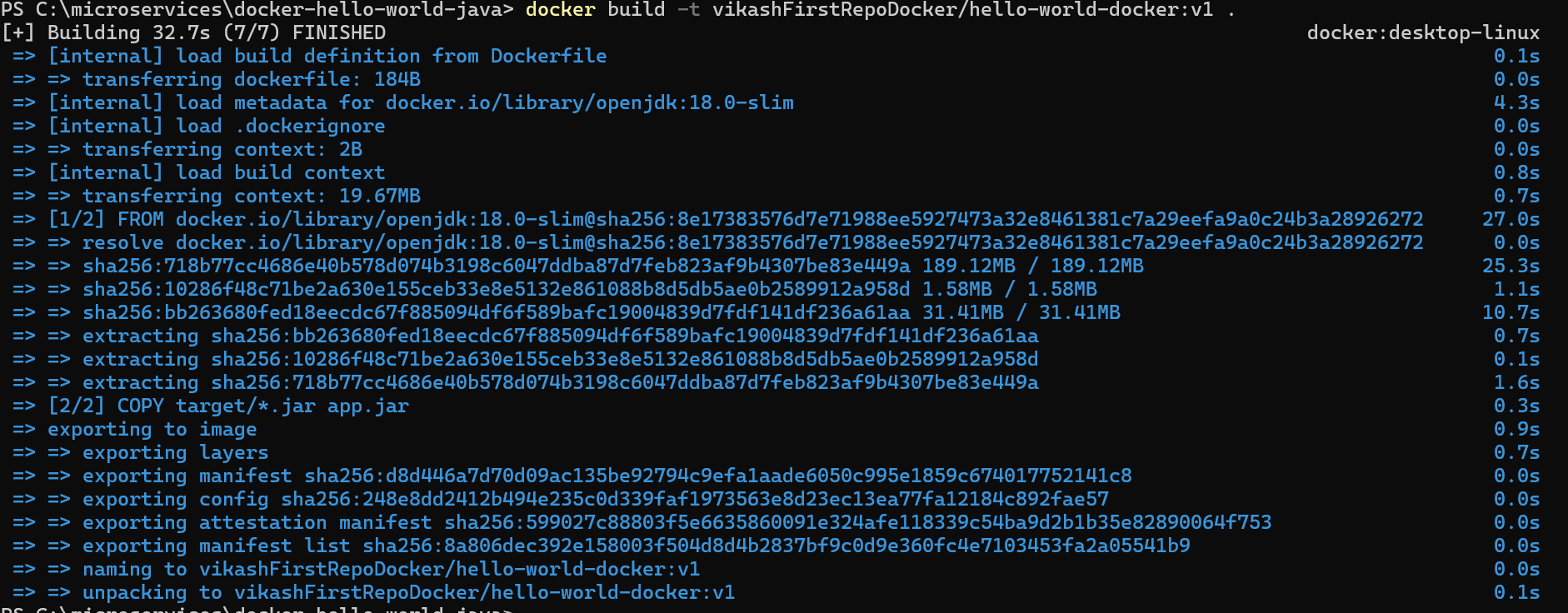
PS C:\microservices\docker-hello-world-java> docker container ls

Response : CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

Step 8 create image by running below command

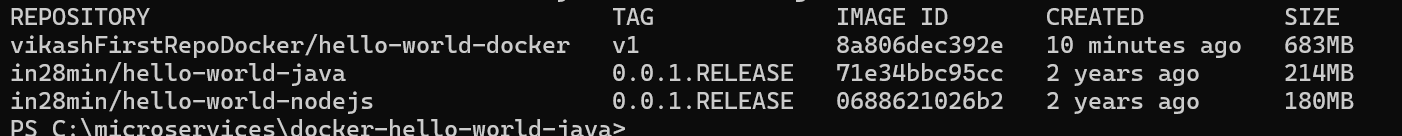
PS C:\microservices\docker-hello-world-java> docker build -t vikashFirstRepoDocker/hello-world-docker:v1 .

Response

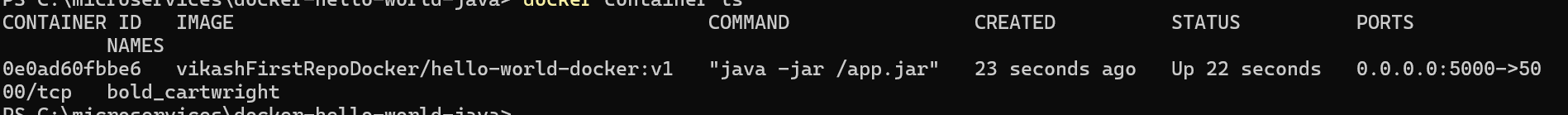


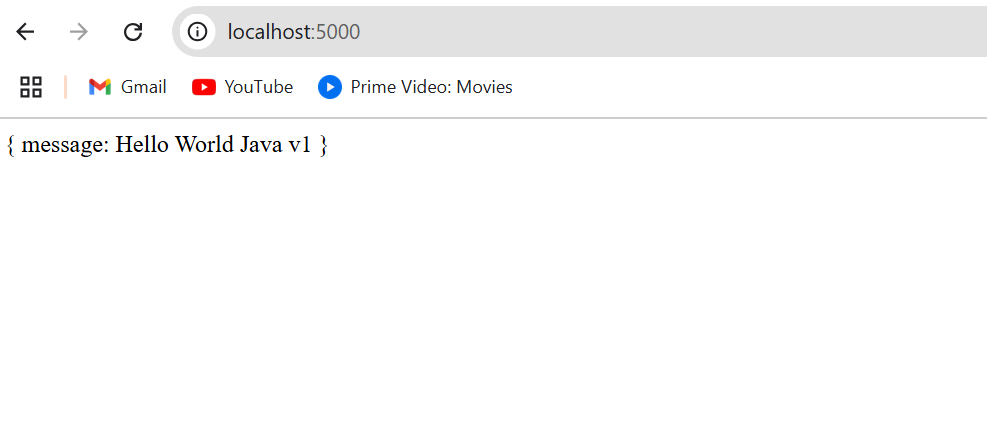
Step 9 run the below command to checlk whether image has been created or not   
  
PS C:\microservices\docker-hello-world-java> docker image list

Response



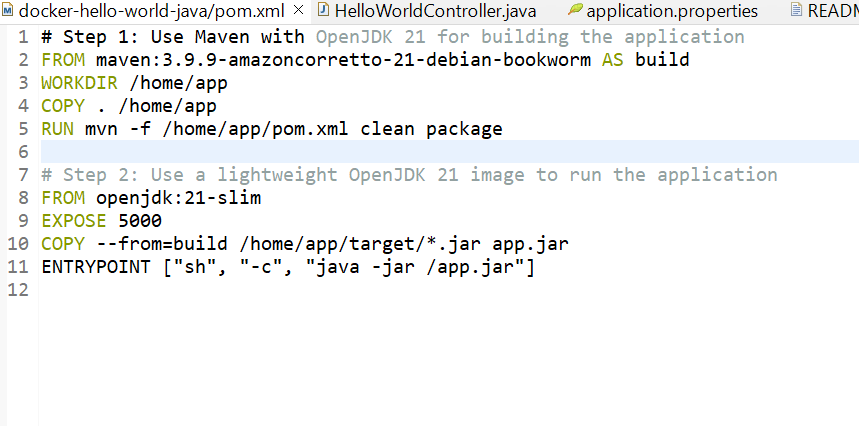
Step 10 Runn this command C:\microservices\docker-hello-world-java> docker container ls  
  
Response





Means we have successfully crated the image

**7.Building Spring Boot Docker Image using Multi Stage Dockerfile**

****

**Explanation of Each Step**

1. **First Stage (build)**:
   * **FROM maven:3.9.9-amazoncorretto-21-debian-bookwormAS build**:
     + This stage uses a Maven image with OpenJDK 21. It provides tools to build the Java application.
   * **WORKDIR /home/app**:
     + Sets the working directory inside the container to /home/app.
   * **COPY . /home/app**:
     + Copies all files from your local project directory into the /home/app directory inside the container.
   * **RUN mvn -f /home/app/pom.xml clean package**:
     + Runs the Maven clean package command to build the application and generate the .jar file in the target directory.
2. **Second Stage (run)**:
   * **FROM openjdk:21-slim**:
     + Uses a lightweight OpenJDK 21 runtime image to run the Java application.
   * **EXPOSE 5000**:
     + Declares that the application will listen on port 5000.
   * **COPY --from=build /home/app/target/\*.jar app.jar**:
     + Copies the .jar file generated in the build stage to the second stage.
   * **ENTRYPOINT ["sh", "-c", "java -jar /app.jar"]**:
     + Defines the command to run the application inside the container using java -jar /app.jar.

**Why Use a Multi-Stage Dockerfile?**

1. **Separation of Concerns**:
   * The first stage (build) includes all dependencies and tools (e.g., Maven) needed to build the project.
   * The second stage (run) contains only the runtime environment, reducing the image size.
2. **Efficient and Clean Runtime Image**:
   * By excluding build tools like Maven from the final image, the container remains lightweight.
3. **Improved Security**:
   * Only the necessary runtime environment is included, minimizing potential vulnerabilities.

**Steps to Build and Run:**

1. **Navigate to the Project Directory**: Ensure you are in the directory where your Dockerfile is located. For your setup:

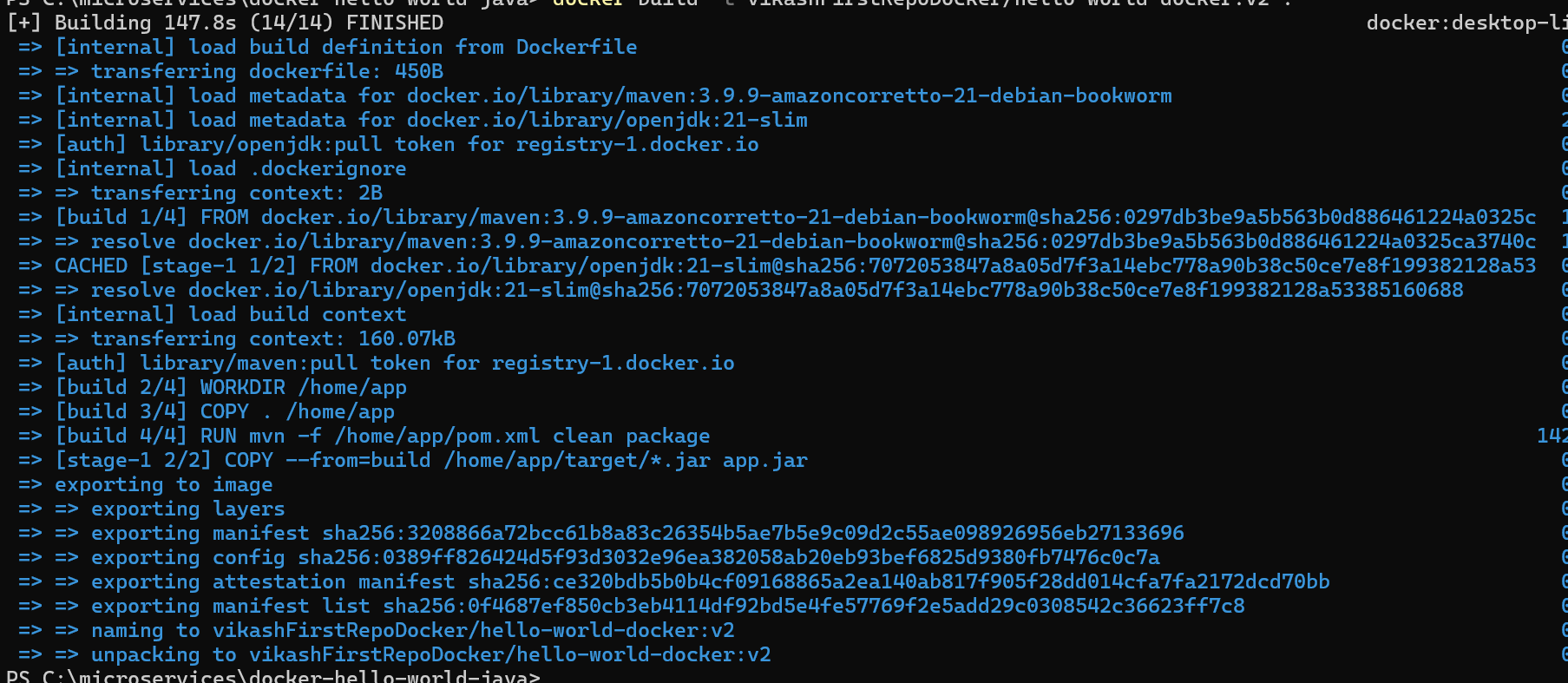
cd C:\microservices\docker-hello-world-java

1. **Run the docker build Command**: Execute the following command to build the Docker image:

docker build -t vikashFirstRepoDocker/hello-world-docker:v2 .

**Expected Output**

When the build completes successfully, you’ll see something like this:

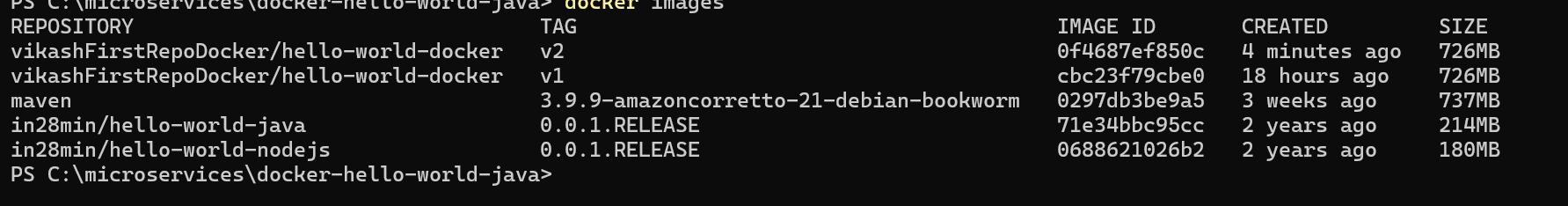


**Verify the Image**

Once built, verify that the image exists in your local Docker repository:

docker images

Expected output:



**Run the Container**

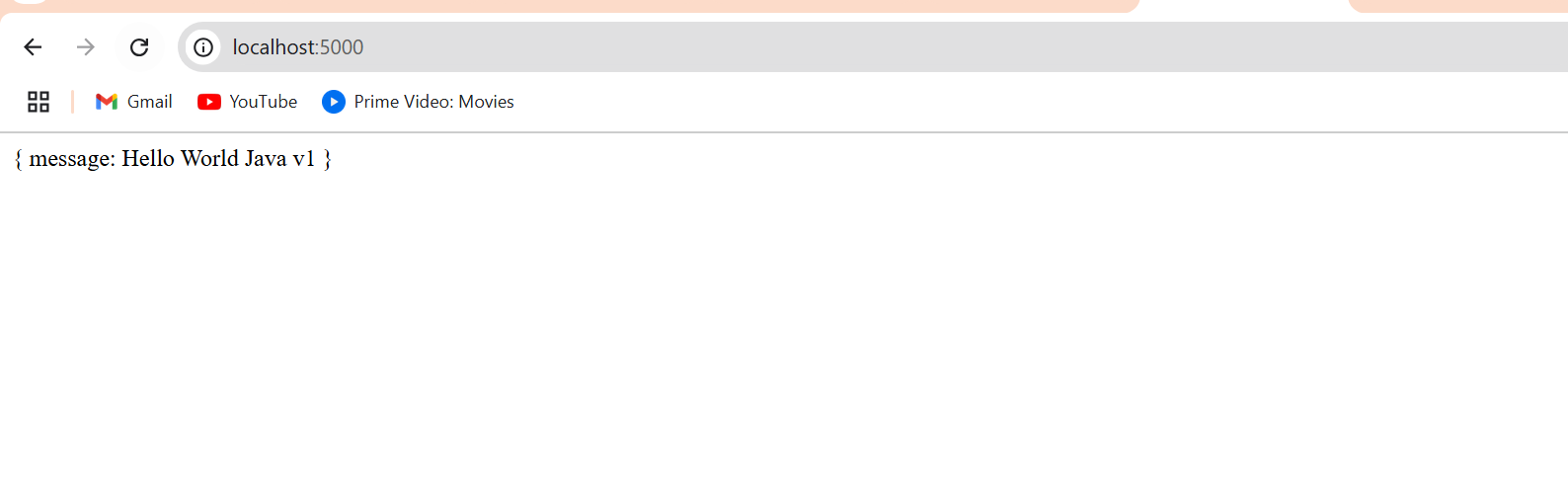
To test the new image, run a container:

docker run -d -p 5000:5000 vikashFirstRepoDocker/hello-world-docker:v2

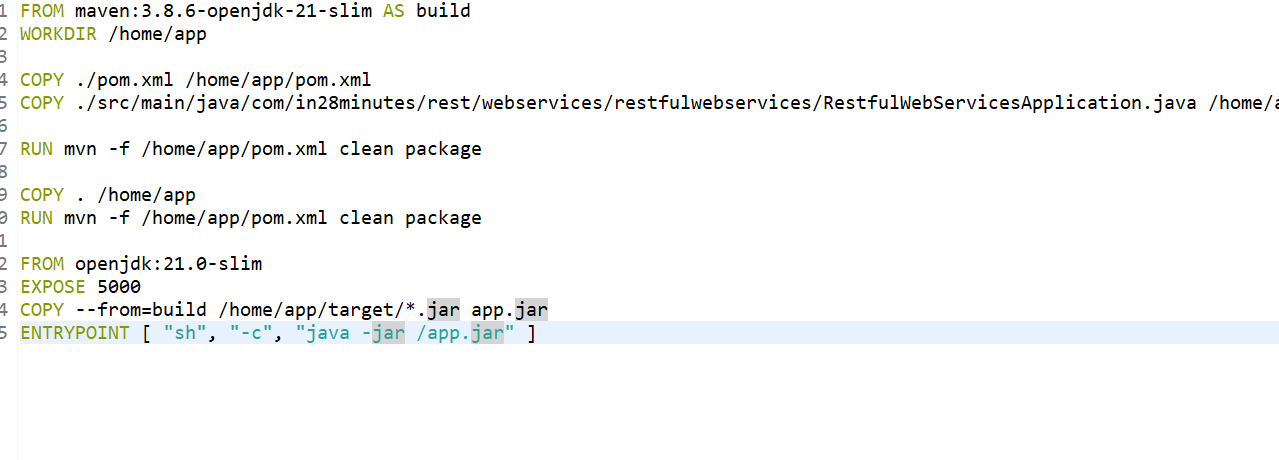
**Test the Application**

Open your browser or use curl to check if the application is running:

<http://localhost:5000>



1. **Building Spring Boot Docker Image - Optimizing Dockerfile**

****

**Copy Specific Java File**

COPY ./src/main/java/com/in28minutes/rest/webservices/restfulwebservices/RestfulWebServicesApplication.java /home/app/src/main/java/com/in28minutes/rest/webservices/restfulwebservices/RestfulWebServicesApplication.java

**Purpose:**  
Copies the main application file separately before copying the rest of the source code.

* **Why?** This creates a separate layer for the application file, allowing it to be cached if the rest of the source code doesn’t change.

FROM maven:3.8.6-openjdk-21-slim AS build

WORKDIR /home/app

COPY ./pom.xml /home/app/pom.xml

COPY ./src/main/java/com/in28minutes/rest/webservices/restfulwebservices/RestfulWebServicesApplication.java /home/app/src/main/java/com/in28minutes/rest/webservices/restfulwebservices/RestfulWebServicesApplication.java

RUN mvn -f /home/app/pom.xml clean package

COPY . /home/app

RUN mvn -f /home/app/pom.xml clean package

FROM openjdk:21-slim

EXPOSE 5000

COPY --from=build /home/app/target/\*.jar app.jar

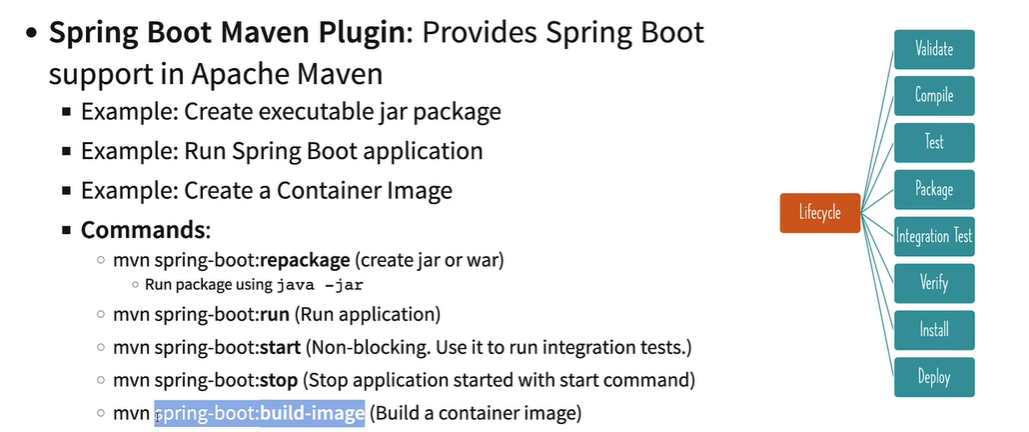
ENTRYPOINT [ "sh", "-c", "java -jar /app.jar" ]

**ummary of the Updated Dockerfile**

1. **Efficient Layer Caching:**
   * By copying pom.xml separately, Maven dependencies are resolved and cached, reducing build times for subsequent changes.
2. **Separate Build Steps:**
   * Separately copying RestfulWebServicesApplication.java creates a specific layer for this frequently updated file.
   * Ensures faster rebuilds when minor changes are made to this file.
3. **Final Source Copy:**
   * Copies all remaining files (COPY . /home/app) after resolving dependencies, ensuring a complete build.
4. **Two Build Runs (mvn clean package):**
   * The first run builds dependencies based on pom.xml.
   * The second run includes all source files and creates the final JAR.
5. **Purpose:**
   * This Dockerfile creates an **optimized, multi-stage build** that leverages **caching** for dependencies and layers to reduce build times and improve efficiency.

Follow the previous commands to run the containers

**8.Building Docker Image with Spring Boot Maven Plugin**



The slide explains the **Spring Boot Maven Plugin**, which is an extension to Apache Maven that simplifies the building and running of Spring Boot applications. Here's a breakdown of the slide:

**Purpose of the Spring Boot Maven Plugin:**

* **Create executable JAR or WAR packages:** Allows you to package your Spring Boot application for deployment.
* **Run Spring Boot applications directly:** Provides commands to run and manage your Spring Boot app.
* **Create container images:** Builds Docker container images directly using Maven.

**Important Commands:**

1. **mvn spring-boot:repackage:**
   * Creates an executable JAR or WAR package.
   * The package can be executed directly using java -jar <jar-file>.
2. **mvn spring-boot:run:**
   * Runs the Spring Boot application directly without needing to package it first.
   * Useful during development.
3. **mvn spring-boot:start:**
   * Starts the Spring Boot application in non-blocking mode.
   * Often used to run integration tests while the application is running.
4. **mvn spring-boot:stop:**
   * Stops an application started with the spring-boot:start command.
   * Useful for cleaning up resources during tests or local development.
5. **mvn spring-boot:build-image:**
   * Builds a container image (e.g., a Docker image) for the Spring Boot application.
   * This is helpful for deploying applications to containerized environments like Kubernetes.

**Lifecycle Phases in Maven (Right Side of the Slide):**

The Maven lifecycle consists of phases, and the plugin integrates well with these phases:

* **Validate:** Validates the project structure and dependencies.
* **Compile:** Compiles the Java source code.
* **Test:** Runs unit tests.
* **Package:** Packages the compiled code into a distributable format (JAR/WAR).
* **Integration Test:** Runs integration tests, typically requiring the app to be started.
* **Verify:** Verifies the package is valid and meets quality standards.
* **Install:** Installs the package into the local Maven repository.
* **Deploy:** Deploys the package to a remote repository for sharing with others.

**Key Takeaway:**

* The Spring Boot Maven Plugin streamlines common tasks such as building, running, and containerizing Spring Boot applications. By using the commands mentioned, you can quickly set up and deploy your applications in various environments.

**9.Introduction to Distributed Tracing**

**Introduction to Distributed Tracing**

**What is Distributed Tracing?** Distributed tracing is a method for monitoring and debugging distributed systems, especially microservices architectures. It helps track requests as they flow through different services in a system, providing end-to-end visibility and identifying bottlenecks or errors.

**Why is Distributed Tracing Needed?**

1. **Complexity in Modern Systems**:
   * In distributed systems, a single user request often spans multiple services and components. Tracing helps identify how each component contributes to the request lifecycle.
2. **Debugging and Troubleshooting**:
   * Distributed tracing pinpoints where delays or errors occur in a system by breaking down the request lifecycle.
3. **Performance Optimization**:
   * Identifies performance bottlenecks across microservices and allows developers to optimize service interactions.
4. **Monitoring SLAs and KPIs**:
   * Helps ensure adherence to Service Level Agreements (SLAs) and other performance metrics.

**How Does Distributed Tracing Work?**

Distributed tracing records and visualizes the flow of requests through a system. Here's how it works:

1. **Trace Context**:
   * Each request is assigned a unique **trace ID**. The trace ID remains consistent across all services involved in fulfilling the request.
   * Each service or operation within the trace is assigned a **span ID** to represent its individual execution.
2. **Spans**:
   * A span represents a unit of work within a trace (e.g., a service call, a database query).
   * Each span contains:
     + Start and end timestamps
     + Operation name
     + Metadata (e.g., service name, error status)
3. **Propagation**:
   * Trace and span IDs are propagated between services using headers (e.g., HTTP headers like traceparent or custom formats).
4. **Collection and Analysis**:
   * Tracing data is collected by a distributed tracing system (e.g., Jaeger, Zipkin, OpenTelemetry).
   * Data is visualized in a trace timeline or graph, showing dependencies and performance metrics.

**Key Components of Distributed Tracing**

1. **Tracing Libraries**:
   * Code libraries embedded in services to instrument and capture trace data (e.g., OpenTelemetry SDKs).
2. **Trace Collectors**:
   * Collects spans and traces from services and aggregates them (e.g., Zipkin, Jaeger).
3. **Visualization Tools**:
   * Provides a user interface for analyzing traces, identifying bottlenecks, and troubleshooting errors.

**Distributed Tracing Tools**

1. **Jaeger**:
   * Open-source tracing tool developed by Uber. Useful for performance monitoring and root cause analysis.
2. **Zipkin**:
   * Another open-source tracing system. Focuses on latency analysis and troubleshooting.
3. **OpenTelemetry**:
   * A unified framework for instrumentation, collecting, and exporting trace data.
4. **Elastic APM, AWS X-Ray, and Datadog**:
   * Managed solutions offering distributed tracing capabilities integrated with monitoring services.

**Advantages of Distributed Tracing**

1. **Improved Visibility**:
   * Provides a clear view of how requests flow across services and where issues arise.
2. **Faster Issue Resolution**:
   * Helps developers quickly identify the root cause of errors or delays.
3. **Enhanced Collaboration**:
   * Teams working on different services can collaborate effectively with shared trace data.
4. **Proactive Performance Monitoring**:
   * Identifies potential bottlenecks before they impact users.

**Challenges of Distributed Tracing**

1. **Overhead**:
   * Additional computational and storage requirements for collecting and processing traces.
2. **Implementation Complexity**:
   * Instrumenting every service with tracing libraries can be time-consuming.
3. **Data Overload**:
   * Managing and analyzing vast amounts of tracing data requires effective filtering and visualization tools.

**Conclusion**

Distributed tracing is an essential practice for modern cloud-native applications and microservices. It provides deep insights into system behavior, helps identify performance bottlenecks, and enables faster debugging and root cause analysis. Tools like OpenTelemetry, Jaeger, and Zipkin make implementing distributed tracing more accessible, ensuring smooth operations and a better user experience in distributed systems.

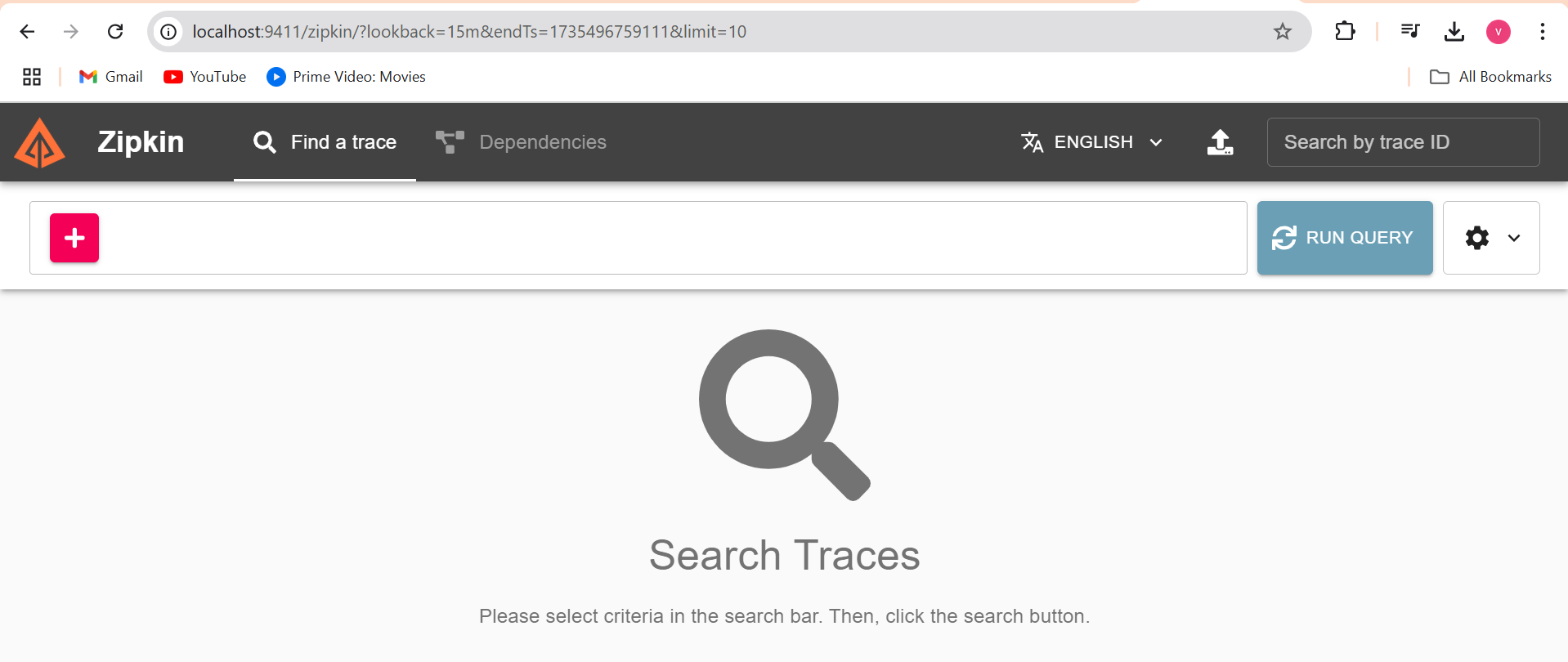
**10. Launching Zipkin Container using Docker**

You can launch **Zipkin** using the following Docker command:

docker run -d -p 9411:9411 openzipkin/zipkin

If you want to use version 2.23.2, the command would be:

docker run -d -p 9411:9411 openzipkin/zipkin:2.23.2



**11. Getting Started with Observability and OpenTelemetry**

**What is Observability?**

Observability is the ability to understand the internal state of a system based on the data it generates. It provides insights into how a system behaves and helps in diagnosing issues, understanding performance, and ensuring reliability. Observability focuses on answering *"Why is this happening?"* when something goes wrong in a distributed or complex system.

**Key Pillars of Observability**

1. **Metrics**:
   * Quantitative measurements of system performance.
   * Examples: CPU usage, memory consumption, request latency, etc.
2. **Logs**:
   * Text-based records of discrete events.
   * Examples: Error logs, audit logs, or debugging statements.
3. **Traces**:
   * Represent the flow of requests across services in a system.
   * Example: A user request travels from the front-end to multiple back-end services.

Observability helps teams monitor applications effectively, improve system performance, detect issues, and enhance the user experience.

**What is OpenTelemetry?**

**OpenTelemetry** is an open-source framework for generating, collecting, processing, and exporting telemetry data (metrics, logs, and traces) from applications and systems to help with observability. It provides a standardized way to instrument your applications, making it easier to integrate with different observability tools.

**Key Features of OpenTelemetry**

1. **Language Support**:
   * Supports popular programming languages like Java, Python, Go, Node.js, C#, and others.
2. **Instrumentation**:
   * Provides libraries and agents to automatically or manually instrument code for telemetry data collection.
3. **Vendor-Neutral**:
   * Allows you to send data to any observability platform (e.g., Prometheus, Zipkin, Jaeger, Datadog, Splunk, etc.).
4. **Unified Model**:
   * Combines metrics, traces, and logs under one framework, making it easier to manage telemetry data.

**Relationship Between Observability and OpenTelemetry**

* **Observability** is the goal: understanding and monitoring the internal state of a system.
* **OpenTelemetry** is the means: a framework that helps collect the necessary data (metrics, logs, traces) for observability.

**Example Usage of OpenTelemetry:**

1. **Traces**:
   * Automatically instrument microservices to generate distributed traces for each user request.
   * Send traces to Zipkin or Jaeger for visualization.
2. **Metrics**:
   * Collect performance metrics like request rates or database query durations.
   * Send metrics to Prometheus for monitoring.
3. **Logs**:
   * Correlate logs with traces for deeper debugging.

**Why Use OpenTelemetry?**

* Standardized instrumentation for observability.
* Easy integration with a wide range of backends and tools.
* Simplifies monitoring for complex systems like microservices architectures.

In short, **OpenTelemetry** is a critical enabler for building effective observability into modern distributed systems.

**12. Connecting Currency Exchange Microservice with Zipkin**

**Brief Overview**

**Micrometer**

* **What it is:** A vendor-neutral facade for application observability in JVM-based systems.
* **Purpose:** Helps instrument your application with **metrics** and **logs** without being tied to a specific vendor.
* **Capabilities:** Supports **metrics**, **logs**, and **tracing** integration.

**OpenTelemetry**

* **What it is:** A framework for generating, collecting, and exporting telemetry data (metrics, logs, and traces).
* **Purpose:** Standardizes observability data across different systems and tools.
* **Use Case:** Enables tracing and metrics for distributed systems and integrates with tools like Zipkin, Jaeger, and Prometheus.

**Zipkin**

* **What it is:** A distributed tracing system.
* **Purpose:** Visualizes and troubleshoots latency in microservice architectures by analyzing traces.
* **Use Case:** Works with OpenTelemetry to display end-to-end trace details for requests.

**Micrometer + OpenTelemetry + Zipkin in Spring Boot**

* **Micrometer** collects **metrics** and **logs** for Spring Boot applications.
* **OpenTelemetry** enables **tracing** for distributed systems and routes telemetry data.
* **Zipkin** displays traces, enabling developers to debug and optimize system performance effectively.

**12. Connecting Currency Exchange Microservice with Zipkin**

Add below dependency in currency-Exchange -service in pom.xml file

<dependency>

<groupId>io.micrometer</groupId>

<artifactId>micrometer-observation</artifactId>

</dependency>

<!-- OPTION 1: Open Telemetry as Bridge (RECOMMENDED) -->

<!-- Open Telemetry

- Simplified Observability (metrics, logs, and traces) -->

<dependency>

<groupId>io.micrometer</groupId>

<artifactId>micrometer-tracing-bridge-otel</artifactId>

</dependency>

<dependency>

<groupId>io.opentelemetry</groupId>

<artifactId>opentelemetry-exporter-zipkin</artifactId>

</dependency>

Add below properties in application .properties

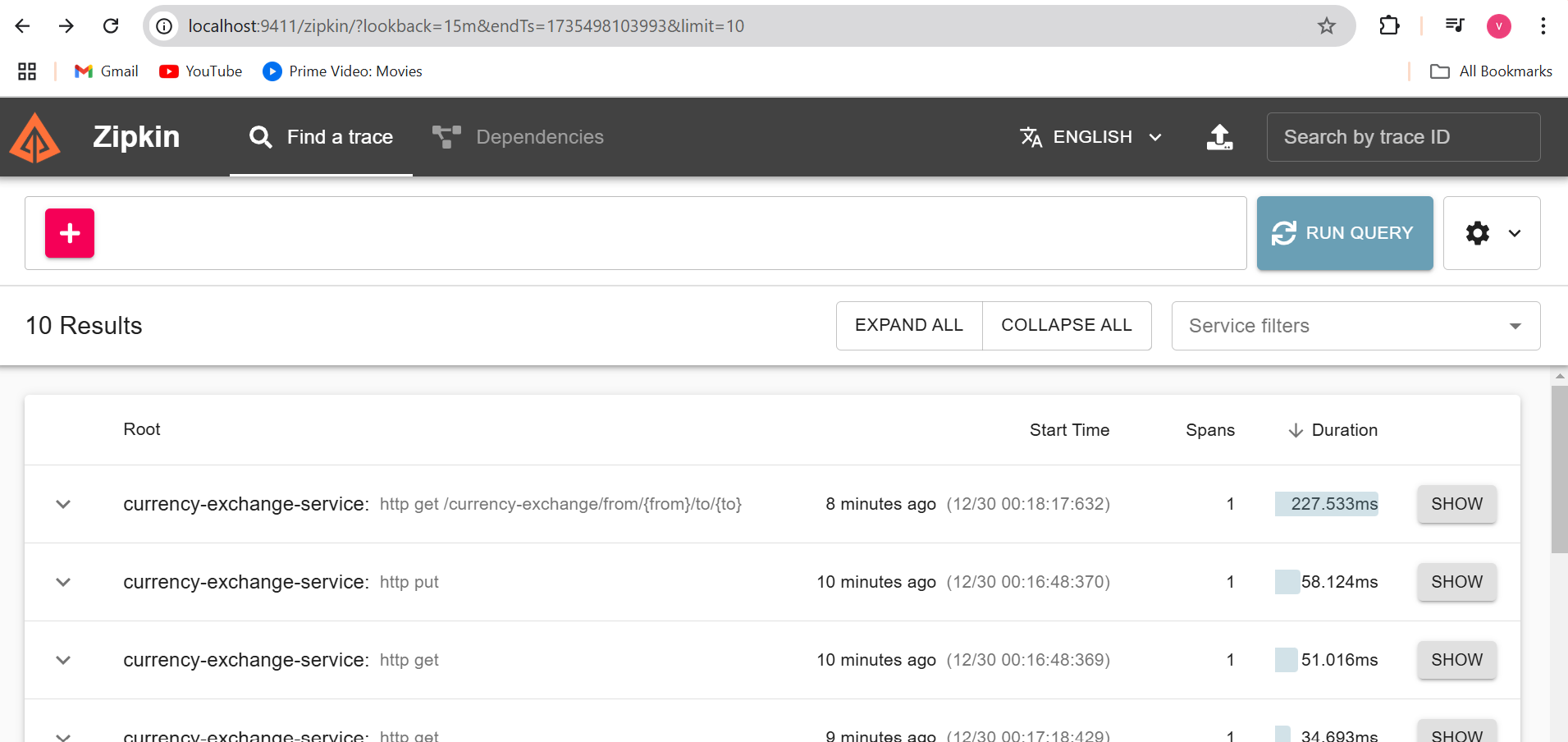
management.tracing.sampling.probability=1.0 #SB3

logging.pattern.level=%5p [${spring.application.name:},%X{traceId:-},%X{spanId:-}] #SB3

Run naming service application and after that currency-exchange service

But before that makesure zipkin is running   
hit this url ‘http://localhost:9411/ “ to check

When both application are up running got to Zipkin dashboard and rin query



**What is Docker Compose?**

Docker Compose is a tool provided by Docker to simplify the process of defining and running multi-container applications. With Docker Compose, you can manage multiple services, networks, and volumes together as a single application.

**Key Features of Docker Compose:**

1. **Single Configuration File:** Use a docker-compose.yml file to define all your services, networks, and volumes.
2. **Multi-Container Management:** Start, stop, and manage multiple containers as a group.
3. **Simplified Commands:** With a single command (docker-compose up), all the services defined in the file are started together.
4. **Portability:** Makes it easy to share your environment setup with others by providing the docker-compose.yml file.
5. **Networking:** Automatically creates a shared network for the defined services to communicate with each other.

**Step-by-Step Guide to Using Docker Compose for Your Projects**

**Steps to Follow if Starting Fresh**

1. **Ensure Docker Is Installed and Running**
   * Verify Docker is installed:

docker -v

* + Verify Docker Compose is installed

docker-compose -v

1. **Prepare Each Service Directory**
   * Ensure each service directory (naming-server, api-gateway, currency-exchange-service, currency-conversion-service) contains:
     + Dockerfile
     + Spring Boot application with dependencies configured.
2. **Create the docker-compose.yml File**

* Place it in the root directory as described above.

1. **Build and Start Services**
   * Use docker-compose build to build images.
   * Use docker-compose up to start services.
2. **Verify Services**
   * Access Naming Server: <http://localhost:8761>
   * Access API Gateway: <http://localhost:8765>
   * Test the services using appropriate endpoints.

bash

Copy code

**1. Modify the docker-compose.yml File**

Based on your application.properties for the Currency Exchange Service, here's the updated docker-compose.yml file:

Docker-compose.yml file

**version: '3.8'**

**services:**

**naming-server:**

**image: naming-server:latest**

**container\_name: naming-server**

**ports:**

**- "8761:8761"**

**api-gateway:**

**image: api-gateway:latest**

**container\_name: api-gateway**

**ports:**

**- "8765:8765"**

**environment:**

**- EUREKA\_SERVER\_URL=http://naming-server:8761/eureka**

**currency-exchange:**

**build:**

**context: ./currency-exchange-service**

**dockerfile: Dockerfile**

**container\_name: currency-exchange**

**ports:**

**- "8000:8000"**

**environment:**

**- SPRING\_DATASOURCE\_URL=jdbc:h2:mem:testdb**

**- EUREKA\_SERVER\_URL=http://naming-server:8761/eureka**

**currency-conversion:**

**build:**

**context: ./currency-conversion-service**

**dockerfile: Dockerfile**

**container\_name: currency-conversion**

**ports:**

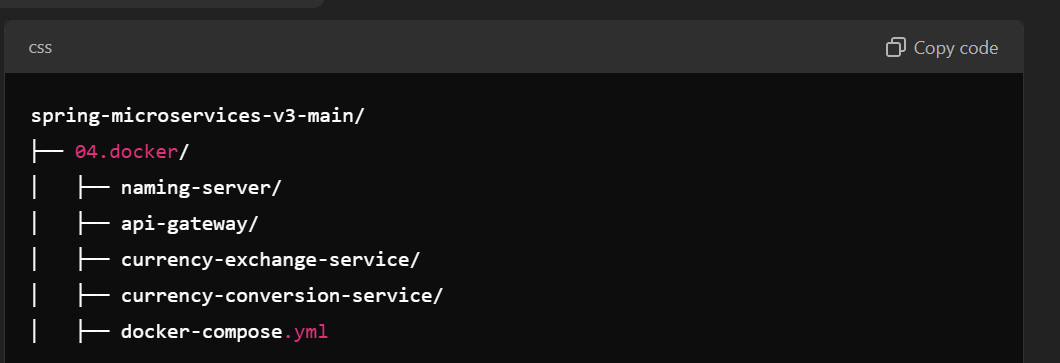
**- "8100:8100"**

**environment:**

**- EUREKA\_SERVER\_URL=http://naming-server:8761/eureka**

**2. Where to Save docker-compose.yml**

* Save the docker-compose.yml file in the root directory where all your projects reside. For your directory structure:



**3. Commands to Build and Run Services**

**Step 1: Build Docker Images Using Docker Compose**

If you want docker-compose to automatically build the images based on your Dockerfile:

bash

Copy code

docker-compose build

This will look for Dockerfile in the respective directories mentioned in the context field of the docker-compose.yml.

Step 2: Start All Services

docker-compose up

Add -d to run in detached mode:

docker-compose up -d

**Step 3: Stop All Services**

To stop the services and clean up:

docker-compose down